O *Fetch* de uma Instrução Escalar por Ciclo Não Inibe o Paralelismo no Nível de Instrução

Christian D. Freitas e Alberto F. de Souza
Departamento de Informática, Universidade Federal do Espírito Santo
Av. Fernando Ferrari, S/N, 29060-970 – Vitória – ES
{christ, alberto}@inf.ufes.br

Resumo—
Máquinas Super Escalares trazem múltiplas instruções escalares do *cache* de instruções por ciclo. Contudo, máquinas que buscam no *cache* de instruções apenas uma instrução escalar por ciclo de relógio têm demonstrado níveis de desempenho comparáveis aos de máquinas Super Escalares, como é o caso de máquinas que seguem a arquitetura *Dynamic Trace Scheduled VLIW* (DTSVLIW). Neste trabalho, mostramos através de experimentos que basta trazer uma instrução escalar por ciclo de máquina do *cache* de instruções para atingir praticamente o mesmo desempenho obtido trazendo várias instruções por ciclo graças à localidade de execução existente nos programas. Fazemos, também, a primeira comparação direta entre as arquiteturas Super Escalar, *Trace Cache* e DTSVLIW. Nossos resultados mostram que uma máquina DTSVLIW capaz de executar até 16 instruções por ciclo tem desempenho 21.9% superior que uma Super Escalar e 6.6% superior que uma *Trace Cache* com hardware equivalente.
Palavras-chave— **Super Escalar, *Trace Cache*, DTSVLIW.**

Abstract—
Superscalar machines fetch multiple scalar instructions per cycle from the instruction cache. However, machines that fetch no more than one instruction per cycle from the instruction cache, such as *Dynamic Trace Scheduled VLIW* (DTSVLIW) machines, have shown performance comparable to that of Superscalars. In this paper we present experiments which show that fetching a single instruction from the instruction cache per cycle allows the same performance achieved fetching multiple instructions per cycle thanks to the execution locality present in programs. We also present the first direct comparison between the Superscalars, Trace Cache and DTSVLIW architectures. Our results show that a DTSVLIW machine capable of executing up to 16 instructions per cycle can perform 21.9% better than a Superscalar and 6.6% better than a Trace Cache with equivalent hardware.
Keywords— **Superscalar, Trace Cache, DTSVLIW.**

I. INTRODUÇÃO

A arquitetura Super Escalar [JOH 91] tem sido extensivamente estudada devido à sua capacidade de explorar o paralelismo no nível de instrução (*Instruction-level Paralelism* – ILP). Atualmente, máquinas implementadas segundo esta arquitetura são capazes de trazer várias instruções escalares do *cache* de instruções por ciclo de máquina. Além disso, estas arquiteturas são capazes de escalonar múltiplas instruções por ciclo de máquina, através de hardware para *dispatch* (despacho) para uma janela de instruções e *issue* (envio) para execução paralela em múltiplas unidades funcionais. Para isto, as máquinas Super Escalares dispõem de barramentos internos largos para realizar o *dispatch* e *issue* de várias instruções por ciclo. Possuem, também, uma grande janela de instruções, utilizada pelo algoritmo de escalonamento (implementado através das ações de *dispatch* e *issue*) para encontrar o ILP disponível nas múltiplas instruções escalares trazidas do *cache*.

A capacidade do estágio de *fetch* de fornecer múltiplas instruções para o estágio de *dispatch* das arquiteturas Super Escalares é um fator importante no desempenho destas arquiteturas. Esta capacidade é limitada pelo gargalo de *fetch*.

O gargalo de *fetch*, também conhecido como gargalo de Flynn [FLY 66], é causado por falhas na predição dos desvios, *cache misses* e *fetches* parciais. No primeiro caso, uma instrução de desvio predita incorretamente altera o fluxo de execução das instruções trazidas do *cache* de instruções, gerando com isso a necessidade de um novo *fetch* e a retirada de instruções despachadas, ou mesmo executadas especulativamente, do *pipeline* [PTS 96] da máquina. No caso de *cache misses*, o estágio de *fetch* fica impossibilitado de fornecer instruções para os estágios de execução até que a instrução alvo do acesso que gerou o *miss* seja trazida da memória. *Fetches* parciais ocorrem devido à organização física dos *caches*: instruções de desvio muitas vezes impedem que todas instruções de um bloco do *cache* de instruções sejam aproveitadas. Quando um desvio é tomado, existe a dificuldade de trazer do *cache* a instrução de desvio e a instrução alvo deste desvio em um mesmo ciclo de máquina.

Graças ao avanço da arquitetura dos *caches* e das técnicas de predição de desvio, nas arquiteturas Super Escalares atuais a ocorrência de *instruction cache misses* e desvios preditos incorretamente podem ter um impacto no desempenho significantemente menor do que o da ocorrência de *fetches* parciais. Por outro lado, os avanços no hardware das máquinas Super Escalares têm possibilitado a construção de máquinas com um caminho mais largo nos estágios do *pipeline*. Em futuras máquinas Super Escalares, capazes de trazer oito ou mais instruções escalares por ciclo de máquina, *fetches* parciais ocorreriam praticamente em todos os ciclos [DES 00b].

O *Trace Cache*, proposto por Rotenberg [ROT 96], é um mecanismo que tem como objetivo aumentar a capacidade do estágio de *fetch* das máquinas Super Escalares atacando as

perdas causadas pelos *fetches* parciais. O princípio básico do funcionamento do *Trace Cache* é armazenar, em uma memória *cache*, as instruções escalares de uma forma contígua, a partir da seqüência em que foram visitadas ou executadas. Isto permite que seja explorada a forte localidade temporal de execução encontrada nos programas [DES 00b]. Com isso, o problema de *fetches* parciais é minimizado e a performance das arquiteturas Super Escalares aumentada.

Existem, contudo, arquiteturas que buscam uma instrução por ciclo de máquina no *cache* de instrução, exploram ILP e não sofrem do problema de gargalo de *fecth*, como é o caso da arquitetura DTSVLIW (*Dynamically Trace Scheduled VLIW*), proposta por de Souza [DES 98].

Uma máquina implementada segundo a arquitetura DTSVLIW busca instruções escalares, uma a uma, do *cache* de instruções e as executa utilizando um processador *pipelined* simples – o *Primary Processor* da arquitetura. Adicionalmente, sua *Scheduling Unit* realiza o escalonamento dinâmico do caminho produzido pela execução destas instruções escalares, montando, com isso, instruções VLIW (*Very Long Instruction Word*) [FIS 84]. Estas instruções VLIW são agrupadas em blocos e armazenadas em um *cache* de instruções VLIW. Se um mesmo trecho de código necessitar ser executado novamente, as instruções deste trecho serão fornecidas pelo *cache* VLIW e executadas por uma *VLIW Engine*, parte da arquitetura DTSVLIW.

A arquitetura DTSVLW é semelhante à arquitetura DIF (*Dynamic Instruction Formatting*) proposta por Nair e Hopkins [NAI 97]. As principais diferenças entre a arquitetura DTSVLIW e a arquitetura DIF são o algoritmo de escalonamento, o mecanismo de renomeação de registradores, a forma de acessar o banco de registradores e a forma de utilização do *cache* VLIW [DES 00a].

Apesar de buscarem instruções escalares uma a uma do *cache* de instruções, o trabalho [DES 00b] mostra que máquinas DTSVLIW possuem um desempenho comparável ao de máquinas Super Escalares. No entanto, os experimentos descritos neste trabalho sofrem dos seguintes problemas: (i) o simulador do processador Super Escalar utilizado executava código de uma *instruction set architecture* (ISA) diferente da do simulador DTSVLIW; (ii) os programas de teste utilizados nos experimentos com cada arquitetura foram os mesmos, mas tinham entradas possivelmente diferentes; e (iii) os programas de teste foram compilados com compiladores diferentes e usando flags de compilação com efeito possivelmente diferente. Neste trabalho apresentamos resultados experimentais mais precisos comparando a arquitetura DTSVLIW e a arquitetura Super Escalar: todos os problemas mencionados da comparação anterior foram removidos. Apresentamos, também, o resultado da primeira comparação da arquitetura DTSVLIW com a arquitetura *Trace Cache*. Os resultados dos nossos experimentos mostram que máquinas implementadas segundo a arquitetura DTSVLIW, capazes de executar até 16 instruções por ciclo, possuem um desempenho 21.9% superior que máquinas Super Escalares e 6.6% superior que máquinas *Trace Cache* com hardware equivalente. Mostramos, ainda, que uma máquina implementada utilizando a arquitetura *Trace Cache* que busca uma instrução escalar por ciclo de máquina no *cache* de instruções apresenta desempenho muito próximo (5.3% inferior) ao de uma máquina que busca um bloco completo do *cache* de instruções por ciclo de máquina. Isto mostra que a característica do código que a DTSVLIW explora – localidade de execução – permite dispensar o escalonamento de mais de uma instrução escalar por ciclo de máquina.

II. Máquinas Super Escalares e Trace Cache.

A. *Máquinas Super Escalares*

O termo Super Escalar foi criado por Agerwala e Cocke [AGE 87] para designar máquinas que realizam o *dispatch* de múltiplas instruções escalares independentes por ciclo de máquina. Para a realização do *dispatch* de múltiplas instruções independentes, uma máquina implementada segundo a arquitetura Super Escalar precisa utilizar um esquema de "olhar adiante" (*look-ahead*) no código [KEL 75].

O termo *look-ahead* deriva de uma classe de esquemas nos quais os programas são especificados de uma maneira serial convencional, mas o processador pode olhar adiante durante a execução e executar instruções escalares tanto em paralelo quanto fora de ordem, desde que não ocorram inconsistências lógicas como resultado desta execução. Se o processador possuir capacidade suficiente, várias instruções podem ser executadas concorrentemente usando *look-ahead* [KEL 75]. O algoritmo de Thornton [THO 64], utilizado no CDC6600, e o algoritmo de Tomasulo [TOM 67], utilizado no IBM 360/91, são esquemas *look-ahead* elaborados para máquinas que despacham apenas uma instrução escalar por ciclo de máquina. Desde a implementação de uma variante do algoritmo de Tomasulo capaz de despachar mais de uma instrução por ciclo em uma máquina comercial, o IBM RISC System/6000 [GRO 90], variações dos algoritmos de Thornton e Tomasulo com esta capacidade vêm sendo utilizadas em várias máquinas, hoje conhecidas como Super Escalares.

A Figura 1 exibe o diagrama de blocos de uma máquina Super Escalar. O núcleo da máquina é o hardware para *look-ahead*, também chamado de hardware de escalonamento dinâmico. A função do hardware de escalonamento dinâmico é despachar instruções decodificadas para a janela de instruções (JI na Figura 1), e selecionar instruções desta janela para enviar para as unidades funcionais (UF na Figura 1) em cada ciclo de máquina. A janela de instruções é de importância fundamental para o hardware de escalonamento dinâmico e pode ser organizada como uma estrutura com muitas entradas para armazenar as instruções escalares e informações para auxiliar o escalonamento, ou como várias estruturas pequenas com conteúdo equivalente, uma para cada unidade funcional.

Durante o processo de *dispatch*, informações de dependência são adicionadas às instruções, as suas saídas são renomeadas, e elas são, então, guardadas na janela de instruções. Durante o processo de *issue*, o hardware de *issue* coleta dados do barramento de resultados, atualiza a janela de instruções com estes dados, seleciona um conjunto de

instruções prontas para serem executadas, e as envia para as unidades funcionais. Os dados coletados do barramento de resultados são guardados na janela de instruções, nas entradas que possuem instruções com um grupo de operandos de entrada incompleto. Quando todos os operandos de entrada de uma instrução estão disponíveis, o hardware de *issue* marca a instrução como pronta para ser executada.

Um dos principais problemas das máquinas Super Escalares altamente paralelas é o hardware de *issue*. Para marcar as instruções como prontas para serem executadas a lógica de *issue* tem que comparar as saídas de todas as unidades funcionais com as entradas de todas as instruções na janela de instruções. Para implementar isto são necessários barramentos para conectar todas as unidades funcionais a todas as entradas da janela de instruções. São necessários, também, vários comparadores e lógica associada proporcional ao número de unidades funcionais vezes o número de instruções que a janela pode guardar. Por isso, processadores Super Escalares com muitas unidades funcionais e grandes janelas de instrução têm um hardware de *issue* complexo, o que pode influenciar negativamente no tempo do ciclo destas máquinas [PAL 97, DES 99a].

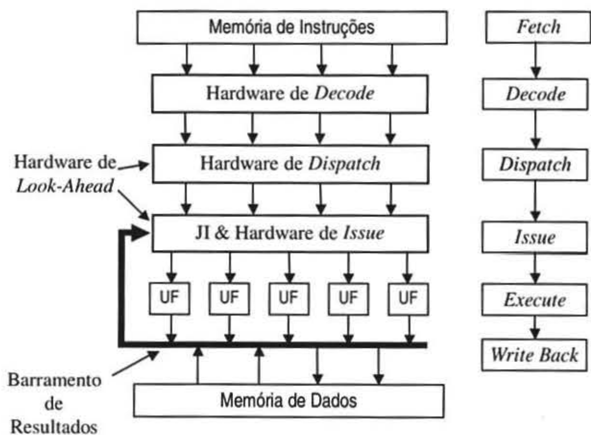

Fig. 1 Máquina Super Escalar

B. Máquinas Super Escalares com Trace Cache

As máquinas Super Escalares atuais já apresentam o problema do gargalo de *fetch*. Estudos mostram que programas inteiros típicos apresentam blocos básicos pequenos, cada um contendo entre quatro e seis instruções na média [YEH 93, CON 95]; ou seja, aproximadamente 19% das instruções destes programas são desvios (condicionais e indiretos) que delimitam blocos básicos. Como estas instruções de desvio são distribuídas de maneira aproximadamente regular ao longo do espaço de endereçamento e aproximadamente 60% delas são desvios tomados [PTS 96 (página 166)], cerca de 12% das instruções executadas em programas inteiros típicos alteram o fluxo de execução (60% de 19%). Por causa disso, uma grande parte dos acessos de *fetch* será pouco eficiente em futuras máquinas Super Escalares com largura de *fetch* de oito instruções ou mais. Estas máquinas Super Escalares terão que trazer instruções de pelo menos um endereço alvo de desvio a cada ciclo de relógio.

Muitos mecanismos foram propostos para melhorar a eficiência do estágio de *fetch* dos processadores baseados nas arquiteturas Super Escalares [CON 95, SEZ 96, YEH 93]. Nestes mecanismos, a cada ciclo de máquina, instruções de regiões não contíguas do *cache* de instruções são buscadas e agrupadas em seqüência utilizando as informações fornecidas por preditores de desvio dinâmicos. Para a realização desta tarefa, o *BTB* (*branch target buffer*) é inspecionado e ponteiros para regiões não contíguas de memória são gerados. É também necessário um *cache* de instruções que permita o acesso e seja capaz de fornecer múltiplos blocos simultaneamente. Estes blocos são alinhados por uma rede de alinhamento, que envia as instruções alinhadas para o estágio de decodificação.

Uma desvantagem destes mecanismos para melhorar a eficiência do estágio de *fetch* é a sua complexidade. Para que estes esquemas funcionem, são necessários poderosos preditores de desvio, *caches* de instruções com várias portas e complexas redes de alinhamento. A arquitetura *Trace Cache*, por sua vez, elimina parte desta complexidade armazenando a seqüência dinâmica de execução das instruções, ao invés das informações para construir esta seqüência [ROT 96]. Na Figura 2 é exibido um diagrama de blocos de uma máquina Super Escalar com *trace cache*, ou máquina *Trace Cache*.

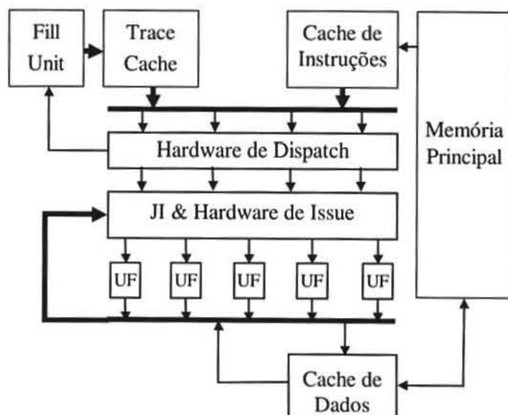

Fig. 2 Máquina *Trace Cache*

Máquinas *Trace Cache* tiram proveito da localidade temporal de execução existente nos programas para explorar o paralelismo no nível de instrução. Uma máquina que emprega esta arquitetura busca instruções do *cache* de instruções e tenta escalonar a execução destas nas suas múltiplas unidades funcionais utilizando, por exemplo, o algoritmo de Tomasulo. Estas instruções são, então, agrupadas por uma *Fill Unit* [MEL 88] na ordem em que elas foram executadas, ao invés da ordem estática determinada pelo compilador, e armazenadas em um *cache* especial chamado *trace cache*. Durante a busca de instruções, o *cache* de instruções e o *trace cache* são examinados ao mesmo tempo. Caso ocorra um *hit* no *trace cache*, um bloco de instruções é fornecido por ele. Este bloco pode conter instruções provenientes de vários blocos do *cache*

de instruções. Graças à localidade de execução, *hits* no *trace cache* são freqüentes, o que minimiza o gargalo de *fetch*.

A arquitetura *Trace Cache* vem sendo alvo de muitas pesquisas devido à sua capacidade de fornecer várias instruções ao estágio de *fetch* dos processadores, minimizando o gargalo de *fetch* e aumentando o desempenho das máquinas Super Escalares [PAT 99, BLA 99, RAK 00]. Contudo, apesar de explorar mais paralelismo do que a arquitetura Super Escalar convencional, arquiteturas Super Escalares melhoradas com *Trace Cache* possuem os mesmos problemas com o hardware de escalonamento dinâmico apresentados pela primeira.

III. MÁQUINAS DTSVLIW

Máquinas implementadas segundo a arquitetura Super Escalar possuem hardware de *dispatch* e *issue* complexo, e possivelmente com um desempenho ruim em termos de tempo de ciclo de máquina. Existem estudos propondo novas arquiteturas que exploram o paralelismo no nível de instrução, cujas máquinas apresentam hardware mais simples do que o hardware de máquinas Super Escalares. A arquitetura DTSVLIW (*Dynamically Trace Scheduled VLIW*) é um exemplo destas novas arquiteturas.

A Figura 3 exibe um diagrama de blocos de uma máquina DTSVLIW. Uma máquina DTSVLIW possui dois *caches* de instruções (o *Instruction Cache* e *o* VLIW *Cache*) e duas unidades de processamento (o *Primary Processor* e a VLIW *Engine*). O *cache* de instruções armazena fragmentos do código original, gerado pelo compilador, enquanto o cache VLIW armazena blocos de instruções VLIW. O código original é executado inicialmente pelo *Primary Processor*. A seqüência de instruções (*trace*) produzida durante esta execução é escalonada pela Unidade de Escalonamento (*Scheduler Unit*) em blocos de instruções VLIW, que são, então, armazenados no *cache* VLIW.

Em uma máquina DTSVLIW, a VLIW *Engine* e o *Primary Processor* nunca operam ao mesmo tempo e não existe a necessidade de transferência do estado entre estas unidades de processamento. Isto é possível devido ao compartilhamento do estado da máquina DTSVLIW entre elas. Desta forma, o desenho das duas unidades é simplificado e é permitido que a VLIW *Engine* compartilhe portas do banco de registradores e o *cache* de dados com o *Primary Processor*. O custo, em ciclos, da transferência da execução entre estas unidades de processamento é igual à soma do número de estágios do *pipeline* de cada uma delas (o conteúdo dos estágios do *pipeline* de uma unidade de processamento deve ser descartado enquanto o conteúdo dos estágios do *pipeline* da outra unidade deve ser restabelecido).

Durante a execução do código no *Primary Processor*, a unidade de *fetch* (*Fetch Unit*) envia endereços diferentes para o *cache* de instruções e para o *cache* VLIW. Para o *cache* de instruções é enviado o contador de programa (*Program Counter* – PC), enquanto que para o *cache* VLIW é enviado o endereço da instrução que estiver no estágio de execução do *Primary Processor* (seta tracejada na Figura 3). Se uma instrução já foi executada antes, pode haver um bloco com o seu endereço no *cache* VLIW. No caso de um *hit* no *cache* VLIW, a VLIW *Engine* assume a execução. Neste momento, o bloco sendo contruído pela Unidade de Escalonamento é enviado para o *cache* VLIW e o conteúdo dos estágios do *pipeline* do *Primary Processor* é descartado, com exceção do estágio *write back*. O PC receberá o endereço de memória que gerou o *hit* no *cache* VLIW e será controlado pela VLIW *Engine* nos ciclos subseqüentes.

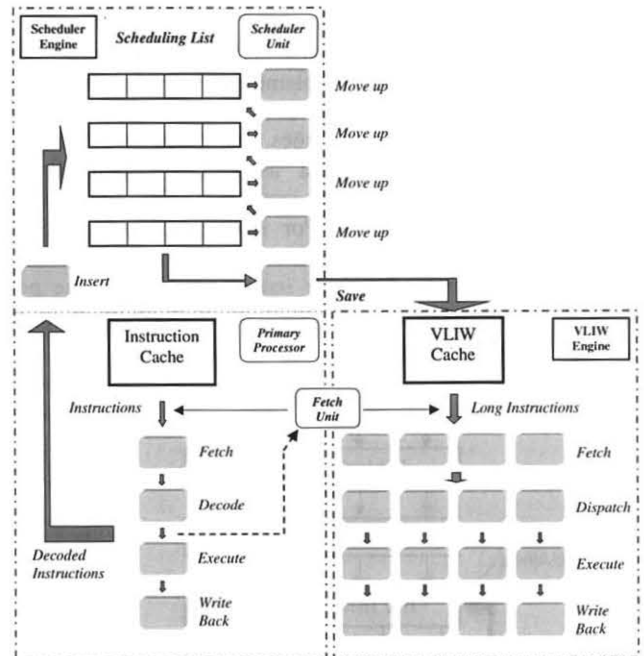

Fig. 3 Máquina DTSVLIW

No caso de um *miss* no *cache* VLIW, o *Primary Processor* assume a execução, realizando o *fetch* no último endereço armazenado em PC, computado pela VLIW *Engine*. A *Fetch Unit* não envia endereços para o *cache* VLIW até que uma instrução chegue ao estágio de execução do *Primary Processor*. Neste momento, a Unidade de Escalonamento reinicia o escalonamento de um novo bloco, cujo endereço é igual ao último endereço produzido pela VLIW *Engine*. Com isto, o último bloco executado pela VLIW *Engine* é encadeado a este que começa a ser escalonado.

A principal motivação para o desenvolvimento da arquitetura DTSVLIW veio da observação de que mesmo *caches* de instruções pequenos (16Kbyte ou 4098 instruções) podem alcançar *hit-rates* mais altas que 99% com os conjuntos de programas de *benchmark* SPEC92 e SPEC95 [GEE 93, CHA 97]. Isto mostra que existe uma grande localidade temporal de execução nos programas. A arquitetura DTSVLIW explora a localidade temporal de execução através do escalonamento do código em blocos de instruções VLIW no primeiro encontro de execução e executando estes blocos na Máquina VLIW em encontros subseqüentes. Experimentos mostram que máquinas DTSVLIW com *caches* de tamanho

compatível com os de *caches* de processadores atuais executam os programas do SPECint95 em modo VLIW durante mais de 90% do tempo, na média, tirando significativo proveito de sua VLIW *Engine* [DES 99b].

TABELA 1
PROGRAMAS BENCHMARK

Benchmark	***Entradas***	***Instruções Executadas***
compress	30000 q 2131	144153036
gcc	-O3 jump.i	176479034
Go	9 9	132169125
ijpeg	Vigo.ppm -GO	220880247
M88ksim	-c < ctl.in	125045424
perl	primes.pl	139264287
Vortex	vortex.in	120451770
Xlisp	queens 7	280939082

TABELA 2
PARÂMETROS FIXOS DA ARQUITETURA SUPER ESCALAR COM TRACE CACHE

Pipeline	• quatro estágios (*fetch, decode, execute* e *write back*) • largura de 16 instruções
Preditor de Desvios	• gshare [MCF 93], history de 15 bits, BHT (Branch History Table) com 32768 contadores saturados de 2 bits • Realiza 3 predições por ciclo de máquina • desvios preditos incorretamente geram uma bolha por 2 ciclos no *pipeline*
BTB	*four way set associative*, 512 *sets*
Latência das Instruções	1 ciclo
RUU (*Register Update Units*)	256
LSQ (*Load/Store Queue*)	32
Trace Cache	*four way set associative*, 8192 *sets*, 16 instruções por entrada com até 3 blocos básicos – 2048Kbyte, com ou sem *atomic blocks*, com ou sem *partial hits*
Cache de Instruções	Perfeito (sem penalidade de miss)
Cache de Dados	Perfeito (sem penalidade de miss)
Número de registradores para *renaming*.	sem limite

TABELA 3
PARÂMETROS DA ARQUITETURA 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
Tamanho da Instrução Decodificada	6 bytes
Latência das Instruções	1 ciclo
Cache VLIW	four way set associative, blocos de 16x16 instruções, 3072-Kbyte
Cache de Instruções	Perfeito (sem penalidade de miss)
Cache de Dados	Perfeito (sem penalidade de miss)
Número de registradores para *renaming*.	sem limite

IV. MÉTODO

Para a realização dos experimentos com a arquitetura Super Escalar descritos aqui, utilizamos o conjunto de ferramentas de simulação simplescalar-3.0 [AUS 97] com a Alpha ISA [DIG 92]. O simplescalar-3.0 possui um simulador Super Escalar *execution-driven* parametrizado, que recebe como entrada código gerado por compiladores ordinários capazes de gerar código para a Alpha ISA. Como o simulador Super Escalar do simplescalar-3.0 não possui *Trace Cache*, implementamos um novo simulador, utilizando o simulador Super Escalar como base, capaz de simular uma máquina *Trace Cache*. Nosso simulador é capaz de simular a arquitetura *Trace Cache* com *partial hits*, *atomic blocks* [PAT 97] ou ambos.

Nos experimentos com a arquitetura DTSVLIW utilizamos uma nova versão do simulador DTSVLIW, implementada utilizando como núcleo o simplescalar-3.0, e capaz de executar código da Alpha ISA.

Em todos os experimentos utilizamos os programas de *benchmark* SPECint95, compilados com o compilador *gcc 2.7.2*, utilizando as flags –O3 –unrolloops. Todos os programas foram executados até o fim. As entradas e o número de instruções executadas são apresentados na Tabela 1.

Os parâmetros que não foram variados nas simulações da arquitetura Super Escalar com *Trace Cache* são apresentados na Tabela 2. Para a arquitetura Super Escalar padrão, utilizamos os mesmos parâmetros da Tabela 2, com exceção dos parâmetros do *Trace Cache* e do preditor de desvios, que é capaz de realizar apenas uma predição por ciclo de máquina.

Na Tabela 3 apresentamos os parâmetros utilizados nas simulações da arquitetura DTSVLIW. As dimensões do *cache* VLIW e do *Trace Cache* foram escolhidas para tornar as duas máquinas equivalentes. Ambos os *caches* são capazes de armazenar o mesmo número de instruções; no entanto, uma instrução tem que estar decodificada para ser guardada no *cache* VLIW da máquina DTSVLIW e possui 6 bytes, enquanto que instruções não decodificadas (4 bytes) podem ser guardadas no *Trace Cache*. Como indicado na Tabela 3, o tamanho dos blocos de instruções VLIW escolhido foi 16x16 (16 instruções VLIW com 16 instruções cada). Este tamanho de bloco permite uma largura de despacho para a DTSVLIW igual à das outras duas máquinas e dá um escopo de escalonamento também igual para todas as máquinas (*scheduling list* DTSVLIW de 256 instruções (16x16) e janela de instruções Super Escalar e *Trace Cache* de 256 instruções). A máquina DTSVLIW simulada incorpora os mecanismos de compactação de bloco descritos em [DES 01].

V. EXPERIMENTOS

A. *Super Escalar x Trace Cache*

O gráfico na Figura 4 mostra os desempenhos das arquiteturas Super Escalar, *Trace Cache* (TC), *Trace Cache* com *partial hits* (TC+PH) e *Trace Cache* com *partial hits* e *atomic blocks* (TC+PH+AT) na execução de cada um dos

programas do SPECint95. A média harmônica (M.H.) dos desempenhos também é mostrada. A arquitetura *Trace Cache* mostra um desempenho 11.9% superior que a Super Escalar na média, a *Trace Cache* com *partial hits* (TC+PH), 13.8%, e a *Trace Cache* com *partial hits* e *atomic blocks* (TC+PH+AT), 14.3%. Estes resultados estão bastante próximos aos resultados apresentados na literatura e confirmam a validade dos modelos das arquiteturas usados. Lembramos, contudo, que estes modelos foram escolhidos para permitir a comparação das arquiteturas em estudo e são bastante otimistas, como pode ser observado pelos parâmetros de simulação da Tabela 2.

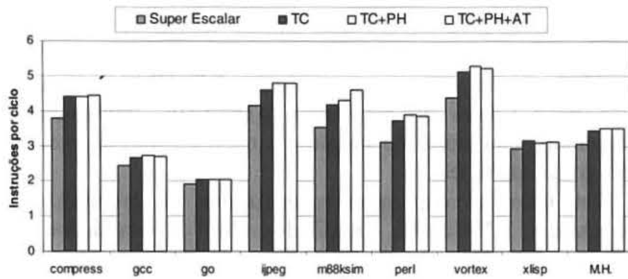

Fig. 4 Desempenho das arquiteturas Super Escalar e *Trace Cache*

B. DTSVLIW x Super Escalar x Trace Cache

A Figura 5 mostra uma comparação entre os desempenhos das arquiteturas Super Escalar, *Trace Cache* (com a melhor configuração discutida na sub-seção anterior), e DTVSLIW. Como o gráfico da Figura 5 mostra, a arquitetura DTSVLIW possui desempenho melhor que o da arquitetura Super Escalar em todos os *benchmarks*, melhor que o da arquitetura *Trace Cache* em gcc, go e vortex, e muito melhor que ambas em xlisp. A DTSVLIW tem um desempenho pior que a *Trace Cache* em compress, ijpeg e m88ksim. Na média, a arquitetura DTSVLIW tem um desempenho 21.9% melhor que a Super Escalar e 6.6% melhor que a *Trace Cache* com *patial hits* e *atomic blocks*.

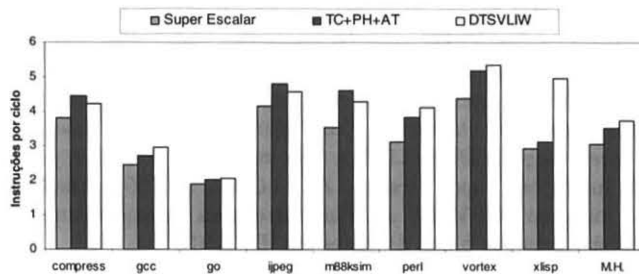

Fig. 5 Desempenho das arquiteturas Super Escalar, *Trace Cache* e DTSVLIW

Nós acreditamos que o desempenho da DTSVLIW é superior que o das arquiteturas Super Escalar e *Trace Cache* por causa do custo dos desvios condicionais preditos incorretamente. Na verdade, a arquitetura DTSVLIW não usa preditores de desvio; no entanto, seu modo de funcionamento resulta no equivalente a um preditor [DES 99a].

Quando em regime, a maior parte da execução DTSVLIW se dá em modo VLIW, já que, quando um desvio segue um caminho diferente daquele seguido durante escalonamento, um novo bloco é escalonado a partir do seu novo alvo. Com isso, quando um desvio segue um caminho diferente daquele seguido durante o escalonamento, a DTSVLIW só perde os ciclos necessários para encher os estágios de *fetch* e *dispatch*, já que muito provavelmente o alvo do desvio é um bloco já escalonado. Por outro lado, a arquitetura Super Escalar tem que descartar instruções em execução especulativa, buscar novas instruções no *cache* de instruções, decodificá-las, escaloná-las (*dispach* e *issue*) e só então executá-las. Embora a arquitetura *Trace Cache* traga instruções do *trace cache* quando em regime, ela tem que fazer o mesmo trabalho de escalonamento que a Super Escalar, com ganho apenas no estágio de *fetch*.

C. Fetch de uma instrução por ciclo

A DTSVLIW traz uma instrução do *cache* de instruções por ciclo de máquina no máximo. Isto não impede que ela alcance desempenho superior que as arquiteturas Super Escalar e *Trace Cache*, como vimos na sub-seção anterior. Com base nisso, decidimos investigar qual seria o desempenho das arquiteturas Super Escalar e *Trace Cache* quando buscando no máximo uma instrução por ciclo do *cache* de instruções. A Figura 6 mostra o resultado dos nossos experimentos.

Como era de se esperar, o desempenho da arquitetura Super Escalar cai para menos que uma instrução por ciclo em todos os *benchmarks*. A arquitetura *Trace Cache*, no entanto, quase não perde desempenho, chegando a ganhar por uma pequena margem das outras em ijpeg. Este resultado comprova a grande localidade de execução presente nos programas: muito pouco tempo, se comparando com o tempo total de execução de cada programa, é gasto produzindo blocos e guardando-os no *trace cache* ou no VLIW *Cache*.

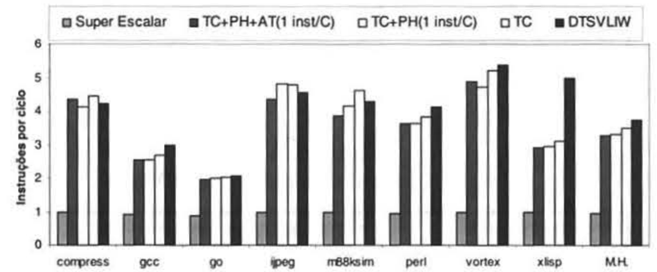

Fig. 6 Desempenho das arquiteturas Super Escalar, *Trace Cache* com *fetch* de uma instrução por ciclo e DTSVLIW

VI. CONCLUSÕES

A arquitetura Super Escalar tem sido usada para implementar a maioria dos processadores de alto desempenho atuais. Contudo, a quantidade de ILP que pode ser explorada por processadores com arquitetura Super Escalar está se

aproximando do limite desta arquitetura. Isso está ocorrendo por causa do gargalo de *fetch* das arquiteturas Super Escalares.

Nos programas inteiros do conjunto de programas de *benchmark* SPEC92, em média 19% das instruções executadas são instruções de desvio [PTS 96 (página 105)] e em média 62% destas mudam a direção do fluxo de execução [PTS 96 (página 166)]. Isto significa que aproximadamente 12% das instruções executadas nestes programas mudam a direção do fluxo de execução – quase uma em oito. Máquinas Super Escalares atuais buscam no *cache* de instruções até quatro instruções a cada ciclo de relógio. Na próxima geração, vai ser possível fazer o *fetch* de oito ou mais instruções a cada ciclo. Isto significa que um número significativo destes acessos de *fetch* trará do cache uma instrução de desvio que mudará a direção do fluxo de execução: muitos destes acessos de *fetch* (na média, quase metade deles para um *fetch* de largura 8) serão apenas parcialmente efetivos. Este problema é conhecido como o gargalo de *fetch* das arquiteturas Super Escalares. Além disso, em vez de simplesmente incrementar o contador de programa para o próximo endereço de *fecth*, máquinas Super Escalares com largura de *fetch* igual a oito teriam de achar o endereço alvo de uma instrução de desvio (possívelmente mais que uma) quase todo ciclo, o que tornaria substancialmente mais complicado o hardware para predição de desvios [CON 95, DES 99a, SEZ 96, YEH 93]. A arquitetura *Trace Cache* tem sido sugerida como uma solução para o problema de gargalo de *fetch* das arquiteturas Super Escalares. Contudo, o *trace cache* e seus dispositivos associados aumentam significativamente a complexidade de implementação, e processadores Super Escalares simples e com altas freqüências de *clock* têm atingido níveis de desempenho mais altos que seus contemporâneos mais complexos [SMI 94, KES 99].

A arquitetura DTSVLIW é uma alternativa às arquiteturas Super Escalar e *Trace Cache* que não sofre do problema de gargalo de *fetch* e pode ser dimensionada para disponibilizar paralelismo de hardware igual ou superior àquele que pode ser extraído por essas duas arquiteturas. Nossos estudos comprovam este fato mostrando que uma maquina DTSVLIW com largura 16 apresenta desempenho 21.9% superior que uma máquina Super Escalar e 6.6% superior que uma máquina *Trace Cache* com hardware equivalente. Nossos estudos mostram também que o *fetch* de uma instrução escalar por ciclo do *cache* de instruções não inibe a exploração do ILP pelas máquinas *Trace Cache*. O desempenho de uma máquina *Trace Cache* que busca uma instrução por ciclo de máquina do cache de instruções e utiliza *partial hits* é apenas 5.3% inferior que uma máquina *Trace Cache* mais complexa, que busca até 16 instruções do *cache* de instruções por ciclo.

REFERÊNCIAS

[AGE 87] AGERWALA, T.; COCKE, J. *High Performance Reduced Instruction Set Processors*, Technical Report RC12434, IBM Thomas J. Watson Research Center, March 1987.

[AUS 97] AUSTIN, T.; BURGER, D. *The SimpleScalar Tool Set*, Technical Report TR-1342, Computer Science Department, University of Wisconsin – Madison, June 1997.

[BLA 99] BLACK, B.; RYCHLIK, B.; SHEN, J. P. *The Block-based Trace Cache*, Proceeding of 26th International Symposium on Computer Architecture, pp. 196-207, 1999.

[CHA 97] CHARNEY, M. J.; PUZAK, T. R. *Prefetching and Memory System Behaviour of the SPEC95 Benchmark Suite*, IBM Journal of Research and Development, Vol. 41, No. 3, pp. 265-285, May 1997.

[CON 95] CONTE, T.; MENEZES, K.; MILLS, P.; PATEL, B. *Optimization of Instruction Fetch Mechanisms for High Issue Rates*, Proc. 22nd Int'l Symp. Computer Architecture, pp. 333-344, June 1995.

[DES 98] DE SOUZA, A. F.; ROUNCE, P. *Dynamically Trace Scheduled VLIW Architectures.* Proceedings of the High-Performance Computing and Networking' 98 – HPCN'98, on Lecture Notes in Computer Science, Vol. 1401, pp. 993-995, April 1998.

[DES 99a] DE SOUZA, A. F. *Integer Performance Evaluation of the Dynamically Trace Scheduled VLIW Architecture.* PhD Thesis, University of London, UK, September 1999.

[DES 99b] DE SOUZA A. F.; ROUNCE P. *Effect of Multicycle Instructions on the Integer Performance of the Dynamically Trace Scheduled VLIW Architecture*, Proceedings of the High-Performance Computing and Networking' 99 – HPCN'99, on Lecture Notes in Computer Science, Vol. 1593, pp. 1203-1206, 1999.

[DES 00a] DE SOUZA, A. F.; ROUNCE, P. *On the Scheduling Algorithm of the Dynamically Trace Scheduled VLIW Architecture.* Proceedings of the International Parallel and Distributed Symposium - IPDPS'2000. Los Alamitos, CA - USA: IEEE Computer Society, 2000. p.565-572.

[DES 00b] DE SOUZA, A. F.; ROUNCE, P. *Dynamically Scheduling VLIW Instructions.* Journal of Parallel and Distributed Computing 60, pp. 1480-1511, December 2000.

[DES 01] DE SOUZA, A. F. *Improving the DTSVLIW Performance via Block Compaction.* Aceito para o 13th Brazilian Symp. on Computer Architecture and High Performance Computing, 2001.

[DIG 92] DIGITAL EQUIPMENT CORPORATION, *Alpha Architecture Handbook*, Digital Equipment Corporation, 1992.

[FIS 84] FISHER, J. A. *The VLIW Machine: A Multiprocessor for Compiling Scientific Code*, IEEE Computer, pp. 45-53, July 1984.

[FLY 66] FLYNN, M. J. *Very High-Speed Computing Systems.* Proccedings of the IEEE, 54, December 1966.

[GEE 93] GEE, J. D.; HILL, M. D.; PNEVMATIKATOS, D. N.; SMITH, A. J., *Cache Performance of the SPEC92 Benchmark Suite*, IEEE Micro, pp. 17-27, August 1993.

[GRO 90] GROHOSKI, G. F. *Machine Organization of the IBM RISC System/6000 Processor*, IBM Journal of Research and Development, Vol. 34, No. 1, pp. 37-58, January 1990.

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

[KEL 75] KELLER, R. M. *Look-Ahead Processors*, ACM Computer Surveys, Vol. 7, No. 8, pp. 177-195, December 1975.

[KES 99] KESSLER, R. E. *The Alpha 21264 Microprocessor*, IEEE Micro, pp. 24-36, March-April 1999.

[PAL 97] PALACHARLA, S.; JOUPPI, N.; SMITH, J. E. *Complexity-Effective Superscalar Processors*, Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 206-218, 1997.

[PAT 97] PATEL, S. J.; FRIENDLY, D. H.; PATT, Y. N. *Critical Issues Regarding the Trace Cache Fetch Mecanism*, Technical Report CSE-TR-335-97, Univ of Michigan, 1997.

[PAT 99] PATEL, S. J.; FRIENDLY, D. H.; PATT, Y. N. *Evaluation of Design Options for the Trace Cache Fetch Mechanism*, IEEE Transactions on Computers, Vol. C-48, No. 2, pp.193-204, 1999.

[PTS 96] PATTERSON, D. A.; HENNESSY, J. L. *Computer Architecture: A Quantitative Approach,* Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[MCF 93] MCFARLING, S. *Combining Branch Predictors*, Digital Western Research Laboratory – WRL Technical Note TN-36, June 1993.

[MEL 88] MELVIN, S.; SHEBANOW, M.; PATT, Y. *Hardware Support for Large Atomic Units in Dynamic Scheduled Machines*, Proceedings of the 21st Annual International Symposium on Microarchitecture, pp. 60-66, December 1988.

[NAI 97] NAIR, R.; HOPKINS, M. E. *Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups*. Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 13-25, 1997.

[RAK 00] RAKVIC, R.; BLACK, B.; SHEN, J. P., *Completion time multiple branch prediction for enhancing trace cache performance*, Proceeding of 26th International Symposium on Computer Architecture, pp. 47-58, 2000.

[ROT 96] ROTENBERG E.; BENNETT, S.; SMITH, J. E. *Trace cache: a low latency approach to high bandwidth instruction fetching*. Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, 1996.

[SEZ 96] SEZNEC, A.; JOURDAN, S.; SAINRAT, P.; MICHAUD, P. *Multiple-Block Ahead Branch Predictors*, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 116-127, 1996.

[SMI 94] SMITH J. E.; WEISS S. *PowerPC601 and Alpha 21064: A Tale of Two RISCs*, IEEE Computer, pp. 46-58, June 1994.

[THO 64] THORNTON, J. E. *Parallel Operation in the Control Data 6600*, Proceedings of the AFIPS Fall Joint Computer Conference, Vol. 26, part 2, pp. 33-40, 1964.

[TOM 67] TOMASULO, R. M. *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal of Research and Development, Vol. 11, No. 1, pp. 25-33, January 1967.

[YEH 93] YEH, T.-Y.; MARR, D. T.; PATT, Y. N. *Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache*, Proc. Seventh Int'l Conf. Super-computing, pp. 67-76, July 1993.