Uma Análise Comparativa entre o Escalonamento de Instruções EPIC e o DTSVLIW

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

Resumo—

Para obter ganhos de desempenho, a arquitetura *Explicitly Parallel Instruction Computing* (EPIC) retira do hardware a responsabilidade de extrair o paralelismo no nível de instrução e a transfere para o compilador, expondo o controle do hardware ao programador do nível convencional de máquina. Já a arquitetura *Dynamically Trace Scheduled VLIW* (DTSVLIW) aposta em um algoritmo simples de escalonamento – implementável em hardware e executado dinamicamente – para obter maiores níveis de paralelismo no nível de instrução e, conseqüentemente, ganhar desempenho. Neste trabalho, nós examinamos três combinações compilador/arquitetura EPIC e as comparamos com uma combinação compilador/DTSVLIW. Nossos experimentos com programas do SPECint95 mostram que, na média, a arquitetura DTSVLIW obtém melhor desempenho porque seu escalonador dinâmico, embora muito mais simples, extrai mais paralelismo que o escalonador do compilador EPIC devido à exploração de informação visível apenas em tempo de execução.

Palavras Chave— **EPIC, DTSVLIW, VLIW**

Abstract—

In order to achieve performance, the *Explicitly Parallel Instruction Computing* (EPIC) architecture takes the responsibility of extracting instruction-level parallelism (ILP) from the hardware and gives it to the compiler. It exposes to the conventional machine level a large part of the hardware control. The *Dynamically Trace Scheduled VLIW* (DTSVLIW), on the other hand, uses a simple scheduling algorithm – hardware implementable and executed dynamically – to exploit ILP and achieve performance. This work examines three compiler/EPIC architecture combinations and compares these with a compiler/DTSVLIW architecture combination. Our experiments show that, on average, the DTSVLIW architecture achieves better performance than EPIC because its dynamic scheduler, although much simpler, harness more ILP due to the exploitation of execution-time information invisible to the EPIC compiler's scheduler.

I. Introdução

Nos últimos anos temos testemunhado uma grande e contínua melhoria no desempenho dos microprocessadores. Esta melhoria foi alcançada, fundamentalmente, sem reescrever programas para uma forma paralela, sem mudar algoritmos ou linguagens e, quase sempre, sem necessidade de recompilação dos programas. Esta melhoria se deve, em parte, à arquitetura Super Escalar [JOH 91] destes microprocessadores, capaz de explorar o paralelismo no nível de instrução encontrado nos programas. Entretanto, para alcançar os níveis atuais de desempenho, um aumento considerável da complexidade do hardware destes processadores tem sido necessário. Esta complexidade pode impor limites para futuros ganhos de desempenho. Na tentativa de resolver este problema foram propostas várias arquiteturas, dentre elas a *Explicitly Parallel Instruction Computing* (EPIC) [GWE 97] e a *Dynamically Trace Scheduled VLIW* (DTSVLIW) [DES 98].

A idéia central por trás das arquiteturas EPIC, uma forma mais complexa de arquitetura *Very Long Instruction Word* (VLIW) [FIS 84], é retirar do hardware a responsabilidade de detectar o paralelismo entre as instruções e passá-la para o compilador, mas dando suporte de hardware para exploração deste paralelismo. Em sistemas EPIC, o compilador é responsável pelo escalonamento do código seqüencial especificado pelo programador em instruções paralelas EPIC.

Já a arquitetura DTSVLIW propõe a execução das instruções de um programa em duas fases: uma seqüencial e outra paralela. Na fase seqüencial, as instruções são trazidas do *cache* de instruções e executadas por um processador *pipelined* simples, ao mesmo tempo em que são escalonadas como instruções VLIW e salvas em blocos de instruções VLIW em um *cache* VLIW. Na fase paralela, as instruções escalonadas durante a fase seqüencial são trazidas do *cache* VLIW e executadas em um processador VLIW.

O principal objetivo deste trabalho é comparar o desempenho de escalonadores complexos, estáticos, implementados no compilador para sistemas EPIC, com o escalonador simples, dinâmico e implementável em hardware da arquitetura DTSVLIW. Para fazer esta comparação, nós estudamos o comportamento de três compiladores EPIC diferentes, compilando para duas *instruction set architectures* (ISA's) EPIC diferentes – HPL-PD [KAT 00] e IA64 [INT 99]. Para o estudo, o código gerado por estes compiladores foi executado em simuladores instrumentados *execution-driven* de máquinas EPIC. As melhores medidas tomadas foram então comparadas com medidas obtidas em nosso simulador DTSVLIW, também *execution-driven*, configurado com hardware equivalente ao das máquinas EPIC. Nossos resultados mostraram que a combinação compilador Intel/ISA IA64 tem desempenho 19,1% melhor que a SGI/IA64, enquanto que a combinação Trimaran/HPL-PD equivalente tem desempenho muito inferior às outras duas. No entanto, esta última com recursos ilimitados de hardware mostrou desempenho 37,7% melhor que a melhor combinação Intel/IA64 na média. Comparando os melhores resultados da arquitetura EPIC com os da DTSVLIW, observamos que esta última pode obter desempenho 37,6% superior que a melhor

combinação compilador EPIC/máquina EPIC viável, na média, e empatar com a combinação compilador EPIC/máquina EPIC com recursos ilimitados. As evidências mostram que a DTSVLIW obtém melhor desempenho porque as informações dinâmicas a respeito da execução dos programas, não disponíveis aos compiladores, permitem que o algoritmo de escalonamento da DTSVLIW, embora muito mais simples, obtenha melhor desempenho que os escalonadores estáticos dos compiladores EPIC.

II. Arquiteturas VLIW e EPIC

Máquinas VLIW são capazes de executar simultaneamente, durante cada ciclo de máquina, diversas instruções procedentes de um mesmo programa. Para isso, elas possuem múltiplas unidades funcionais que podem operar em paralelo. Um bloco de instruções escalares que pode ser executado em paralelo define uma *instrução longa* (termo usado neste trabalho como sinônimo de instrução VLIW). Uma instrução longa especifica quais instruções devem ser executadas, e em quais unidades funcionais da máquina, e usualmente tem tamanho (número de *slots* para instruções escalares) fixo. Se não existe uma quantidade suficiente de instruções que possam ser executadas em paralelo para preencher os *slots* de uma instrução longa, estes serão preenchidos com instruções NOP (*no operation*). Se *n* for o número de *slots*, então *n* instruções por vez serão trazidas do *cache* de instruções para execução. Estas instruções são enviadas diretamente para execução: o hardware não verifica se as instruções que fazem parte da instrução longa podem ser executadas em paralelo. É tarefa do compilador detectar o paralelismo e agrupar as instruções do programa na forma de instruções longas.

A ausência de hardware para detecção do paralelismo e orquestração da execução paralela de instruções torna máquinas implementadas segundo a arquitetura VLIW simples e capazes de operar em altas freqüências de *clock*. No entanto, um programa compilado para uma máquina VLIW não executa em uma outra máquina VLIW com instrução longa de tamanho diferente, mesmo que seja da mesma família. Para executar corretamente, o programa precisaria ser recompilado. Este problema é conhecido como o problema de compatibilidade de código VLIW.

A arquitetura EPIC é considerada uma evolução da arquitetura VLIW e não sofre do problema de compatibilidade de código. Como na arquitetura VLIW, é capaz de executar mais de uma instrução em um mesmo ciclo de máquina, possuindo também unidades funcionais que operam em paralelo. Também pesa sobre o compilador a responsabilidade de detectar o paralelismo no nível de instrução. No entanto, a arquitetura EPIC se diferencia da arquitetura VLIW porque, nesta última, a ligação de uma instrução com o hardware que vai executá-la é feita pelo compilador, enquanto que na primeira isto é feito pelo hardware. Isto permite contornar o problema de compatibilidade de código, mantendo a maioria das características positivas das máquinas VLIW.

A. Escalonamento estático

Com a transferência da responsabilidade de detectar e explorar o paralelismo no nível de instrução para o compilador, este se tornou vital na obtenção de desempenho em sistemas VLIW e EPIC. Um compilador VLIW/EPIC detecta e explora o paralelismo através da ação de escalonamento. A essência do escalonamento está em reordenar o código seqüencial fazendo uso racional dos recursos de hardware com o objetivo de minimizar o tempo de execução do programa. O escalonamento em compiladores VLIW/EPIC pode ser feito abrangendo todo o programa e opera sobre o código gerado após as otimizações tradicionais, independentes de máquina, feitas pelo compilador (*loop invariant motion*, *common subexpression elimination*, *induction variable simplification*, *inline*, *loop unrolling*, etc. [AHO 86]).

Para permitir o escalonamento, as instruções são em geral agrupadas em blocos básicos (bloco de instruções com entrada apenas no início e sem instruções de desvio, exceto no final). Um grafo é criado com os blocos básicos como vértices e os fluxos de controle possíveis como arestas. O tamanho de um bloco básico é em geral pequeno – de 5 a 20 instruções na média [PAT 96] – fazendo com que a quantidade de paralelismo no nível de instrução existente dentro de um bloco básico seja bastante limitada. *Trace Scheduling* [FIS 81] é uma técnica para exploração do paralelismo além dos limites dos blocos básicos. Nesta técnica, utilizam-se heurísticas ou instrumentação (*profiling*), para selecionar o caminho (*trace*) com maior freqüência de execução no grafo. Os blocos básicos pertencentes ao caminho selecionado formam um novo e único bloco. As instruções são tratadas como sendo de um bloco básico, com pouca atenção dada às instruções de desvio, exceto que elas devem permanecer na ordem original. Cada instrução do bloco é, então, escalonada, utilizando o algoritmo de *List Scheduling* [FIS 81, ADA 74], podendo ser movimentada para cima ou para baixo dos limites originais de seu bloco básico. O escalonamento sem considerar as instruções de desvio pode levar a inconsistências quando o fluxo de controle deixa o *trace* ou quando desvios são tomados para dentro do *trace*. A inserção de código de reparo (*bookkeeping*) em todos os pontos de entrada e saída do trace é necessária para manter a semântica do programa. Após escalonar o *trace* mais executado, o compilador seleciona e escalona o segundo, depois o terceiro e assim sucessivamente até o esgotamento dos *traces*.

Na técnica de escalonamento conhecida como *superblock scheduling* são formados superblocos [HWU 93] para aliviar as operações de *bookkeeping*. Um superbloco é um trace que não tem entradas laterais (*side entrances*), isto é, o controle só pode entrar no topo do *trace*, mas pode sair em um ou mais pontos. Um trace comum pode ser convertido em um superbloco pela eliminação de suas *side entrances*. Uma *side entrance* pode ser eliminada copiando as instruções do seu ponto de entrada ao fim do *trace* (*tail duplication*) e redirecionando a *side entrance* para o novo *trace* formado.

Uma técnica similar ao *superblock scheduling* é o *hyperblock scheduling* [MAH 92]. Assim como um superbloco, um hiperbloco é um trace onde o controle só pode entrar no topo do *trace*, mas pode sair em um ou mais pontos. Entretanto, diferente de um superbloco, instruções predicadas [PAR 91] podem ser utilizadas dentro de um hiperbloco. Desta forma, um hiperbloco pode conter instruções pertencentes a mais de um caminho de controle.

Trace scheduling, *superblock scheduling* e *hyperblock scheduling* são técnicas de escalonamento global de código que podem ser aplicadas em trechos acíclicos do programa. Apesar da possibilidade de aplicá-las a trechos cíclicos (*loops*), estas técnicas não exploram com eficiência o paralelismo no nível de instrução existente em *loops*.

Software Pipelining [ALL 95] é uma técnica para explorar o paralelismo no nível de instrução presente em *loops*. É o equivalente em software ao *pipeline* do hardware [RAM 77]. Em uma máquina *pipelined*, várias instruções em diferentes fases de execução podem ocupar diferentes estágios do *pipeline* da máquina, enquanto que em um *software pipelined loop*, várias instruções pertencentes a diferentes iterações do *loop* podem coexistir em uma mesma seqüência de instruções. Um compilador implementando *Software Pipelining scheduling* deve intercalar instruções pertencentes a diferentes iterações de um *loop* de forma tal que o paralelismo no nível de instrução existente no *loop* seja exposto. Outra importante técnica de escalonamento de *loops* é *loop unrolling* [RAU 93].

B. A ISA EPIC HPL-PD e a ISA EPIC IA64

Em 1994, a HP publicou a especificação da máquina EPIC *HP Laboratories PlayDoh* (HPL-PD) [KAT 94]. A HPL-PD é uma arquitetura EPIC experimental genérica de 32 bits parametrizável. Na arquitetura HPL-PD, quase todas as instruções podem ser predicadas. Além disso, é permitido especificar na instrução *load* o nível de *cache* esperado para leitura, e especificar na instrução *store* o nível de *cache* onde o dado deve ser gravado. A arquitetura HPL-PD permite, também, a implementação de desvios condicionais utilizando três instruções: *prepare-to-branch* que calcula o endereço alvo; *compare*, que resolve a condição de desvio; e *branch*, que executa efetivamente o desvio. Isto permite ao compilador, estaticamente, escalonar instruções *prepare-to-branch* e *compare* para executarem em paralelo com outras instruções antes da instrução de *branch* relacionada. Instruções *prepare-to-branch* informam ao hardware sobre a possibilidade e probabilidade (predição estática) de um desvio no fluxo de controle, fazendo com que, tipicamente, seja iniciado a busca de instruções (*prefetch*) a partir do endereço alvo calculado.

Como resultado de uma aliança entre a HP e Intel, foi especificada a ISA IA64 [INT 99], de 64 bits, trazendo características da HPL-PD instanciadas em uma ISA EPIC específica. A ISA IA64 provê as mesmas facilidades citadas anteriormente da ISA HPL-PD. Contudo, na IA64, as instruções são sempre agrupadas em uma estrutura alinhada de 128 bits chamada *bundle*. Cada *bundle* possui três *slots* para instruções de 41 bits e um campo *template* de 5 bits. O formato de um *bundle* é mostrado na Figura 1.

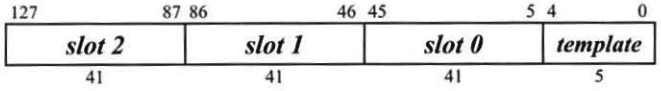

Fig. 1: Formato de um *bundle*

As três instruções de um *bundle* podem executar em paralelo a menos que *stops* sejam inseridos pelo compilador. *Stops* são inseridos para indicar que uma ou mais instruções antes do *stop* têm algum tipo de dependência com uma ou mais instruções após o *stop* e devem ser executadas em ciclos separados. A posição dos *stops* dentro dos *bundles* é especificada no campo de *template*, que também especifica o mapeamento dos *slots* de instrução com os respectivos tipos de unidades funcionais onde elas serão executadas. Nem todos os mapeamentos possíveis estão disponíveis.

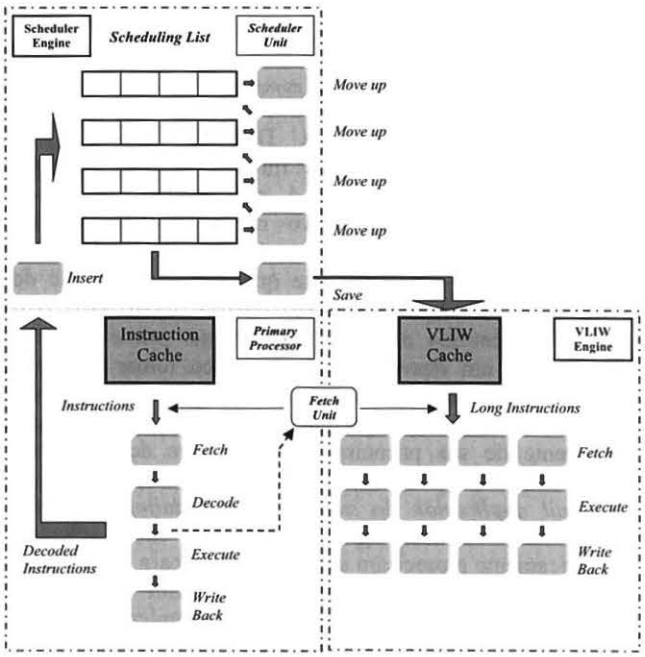

Fig. 2 A Arquitetura DTSVLIW

Máquinas que implementam a ISA IA64 podem enviar para execução mais de um *bundle* por ciclo. No entanto, é responsabilidade do hardware despachar as instruções nos *bundles* para as unidades funcionais apropriadas e cuidar para que sua execução paralela não fira o escalonamento feito pelo compilador. Assim, ao mesmo tempo em que esta ISA tira proveito de características VLIW e permite compatibilidade de código, ela retém parte da complexidade do hardware de *issue* Super Escalar [JOH 90].

III. ARQUITETURA DTSVLIW

Outra solução para o problema de compatibilidade de código VLIW é o conceito DIF (*Dynamic Instruction Formatting*) proposto por Nair e Hopkins [NAI 97]. A arquitetura *Dynamically Trace Scheduled* VLIW (DTSVLIW) [DES 98] segue o conceito DIF alcançando, no entanto, um desempenho semelhante, ou melhor, que o da arquitetura DIF. Além disso, máquinas DTSVLIW são mais simples que máquinas DIF e possivelmente muito mais fáceis de implementar [DES 00].

A Figura 2 mostra um diagrama de blocos da arquitetura DTSVLIW. Em um processador DTSVLIW, a Máquina Escalonadora (*Scheduler Engine*) traz instruções do cache de instruções (*Instruction Cache*) e as executa pela primeira vez usando um processador *pipelined* simples – o processador primário (*Primary Processor*). Além disso, sua unidade de

escalonamento (*Scheduler Unit*) escalona dinamicamente a seqüência de instruções (*trace*) produzida durante a execução no processador primário em instruções VLIW, agrupa estas instruções VLIW em blocos e salva estes blocos no *cache* VLIW (VLIW *Cache*). Se o mesmo código é executado novamente, ele é trazido do *cache* VLIW pela máquina VLIW (VLIW *Engine*) e executado em modo VLIW. Em um processador DTSVLIW, a máquina escalonadora provê compatibilidade de código objeto e a máquina VLIW provê desempenho e simplicidade VLIW.

A. Escalonamento dinâmico DTSVLIW

A *Scheduling Unit* da DTSVLIW faz *superblock scheduling* dinamicamente. O fluxo de instruções produzido pelo processador primário é escalonado em blocos de instruções longas que são salvos no *cache* VLIW. Cada bloco pode conter um ou mais blocos básicos. O escalonamento é feito de forma a permitir que qualquer instrução de desvio dentro do bloco possa sair sem efeitos colaterais e o único ponto de entrada de cada bloco é sua primeira instrução. Assim, se um desvio dentro do bloco tomar um caminho diferente daquele que foi tomado durante o escalonamento e levar a uma instrução pertencente a um bloco existente mas diferente de sua primeira instrução, este desvio causará o escalonamento de um novo bloco. Esta operação é equivalente ao *tail duplication* do *superblock scheduling*. Contudo, no *superblock scheduling* os *traces* são selecionados estaticamente e precisam ser apropriados para todo o conjunto de entradas possíveis do programa. Em contraste, uma máquina DTSVLIW seleciona *traces* dinamicamente, obtendo desempenho independente da entrada do programa.

A principal operação executada pela DTSVLIW é a *move up*, a qual movimenta instruções, vindas do *Primary Processor*, através da *Scheduling List* da máquina (Figura 3) para produzir instruções longas. Várias operações *move up* podem ocorrer em paralelo em um único ciclo de máquina de forma *pipelined*, limitado a uma instrução por instrução longa. Esta instrução é dita uma instrução candidata, ou *candidate instruction*.

Uma instrução fornecida pelo *Primary Processor* é colocada no final da lista de instruções longas. Nos ciclos subseqüentes esta instrução pode sofrer a operação de *move up* se ela não tiver alcançado o início da lista, se existir espaço na instrução longa no nível superior, e se não existir dependência com as instruções da instrução longa no nível superior. 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*. Abaixo mostramos um exemplo da operação *install*:

Sub r1, r2, r3		install ⇒	sub r1, r2, r3	
Add r3, r4, r5			add r3, r4, r5	

Register renaming torna possível o escalonamento de instruções mesmo na presença de dependência entre as instruções, exceto dependência de dados verdadeira. O Algoritmo de escalonamento da DTSVLIW usa a operação *split* para os casos de dependência de controle, de saída e anti-dependência. 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. Abaixo mostramos um exemplo da operação *split*:

Sub r1, r2, r3		split ⇒	sub r1, r2, r3	add r4, r5, r32
Beq r3, 1000	add r4, r5, r6		beq r3, 1000	COPY r32, r6

Desvios condicionais e indiretos não podem ser *moved up* ou *split*. Eles são *installed* quando inseridos na *Scheduling List* e estabelecem uma *tag* para a instrução longa. Todas as instruções *installed* subseqüentemente 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 *branch* seguir a mesma direção obedecida durante o seu escalonamento).

Quando não existir mais espaço na *Scheduling List* para novas instruções, seu conteúdo é salvo no VLIW *Cache* como um bloco, uma instrução longa por ciclo. No entanto, instruções podem continuar sendo inseridas na *Scheduling List* ao mesmo tempo em que o bloco é salvo [DES 00].

Instruções de leitura e escrita na memória também podem ser *split*, o que pode causar *memory aliasing* [FIS 84] e exceções. O modo como a DTSVLIW trata destes casos, escalona instruções que executam em mais de um ciclo, entre outros detalhes da arquitetura, está descrito em [DES 00].

IV. Uma comparação entre o escalonamento estático EPIC e o dinâmico DTSVLIW

Para comparar o escalonamento estático EPIC com o dinâmico DTSVLIW, utilizamos os simuladores EPIC SKI [HPL 00] e Trimaran [HPL 98], e nosso simulador DTSVLIW.

O simulador SKI interpreta a ISA IA64. Ao final da execução de um programa, o simulador apresenta o número de instruções executadas (inclusive NOP's) e o número de *stops* encontrados na execução do programa. O simulador considera unitária a latência de todas as instruções, e que não existe limitação de recursos (quantidade de *slots* em uma instrução longa e número de unidades funcionais). Desta forma, a quantidade de *stops* (*cycles*, no simulador) informada pelo simulador SKI representa o menor número de ciclos que seriam gastos na execução do programa sendo interpretado. Em uma implementação em silício, a latência não unitária das instruções, *cache misses*, *branch mispredictions*, etc, imporia uma quantidade de ciclos muito maior. No entanto, esta medida expõe o desempenho do escalonador e, por isso, é usada aqui para este fim.

O simulador do ambiente Trimaran interpreta o conjunto de instruções definidas na especificação HPL-PD [KAT 95]. O ambiente Trimaran é parametrizável quanto ao número de registradores, quantidade de unidades funcionais, latência das instruções, etc.

O simulador DTSVLIW interpreta a ISA Alpha [DIG 92]. Ele aceita como entrada quaisquer programas compilados para o sistema operacional OSF-1 e os executa fielmente segundo o modelo da arquitetura DTSVLIW descrito na Seção III. A máquina DTSVLIW simulada incorpora, também, os mecanismos de compactação de bloco descritos em [DES 01]. O simulador é parametrizável quanto ao número de registradores, quantidade de unidades funcionais, latência das instruções, etc.

Todos os simuladores interpretam apenas o código que executa em modo usuário, incluindo as instruções que fazem parte de bibliotecas que foram ligadas ao código do programa. Instruções que rodam em modo privilegiado (instruções do sistema operacional) não são interpretadas nem contabilizadas.

TABELA 1
PROGRAMAS E ENTRADAS UTILIZADAS

Programas SPECInt95	Entradas
099.go	9 9
124.m88ksim	dcrand.lit
129.compress95	30000 q 2131
130.li	queens 7
132.ijpeg	vigo.ppm.fast -GO
134.perl	Primes.pl
147.vortex	vortex.in

Em todos os experimentos foram utilizados os programas do SPECInt95 (exceto gcc, por não ser 64 bits compatível e, portanto, ainda não compilável para ISA IA64). Os programas e as entradas utilizadas estão listados na Tabela 1. Os códigos executáveis EPIC foram gerados pelos compiladores: IA64 Itanium™ SGI PRO64 C++ Compiler versão 0.13, o IA64 Itanium™ Intel C++ Compiler 5.0.1 for Linux beta version build 20010418, e o HPL-PD Trimaran 2.0 Compiler. Os executáveis para a ISA Alpha foram gerados pelo compilador gcc 2.7.2.

A. Parâmetros Utilizados

Para o compilador Intel, a *flag* de otimização –O2 habilita todas as otimizações clássicas (*global register allocation, register variable detection, common subexpression elimination, etc*) e *software pipelining*. Embora não seja mencionado na documentação do compilador, acreditamos que o mesmo usa *trace scheduling* ou alguma de suas variações (*hyperblock scheduling*, por exemplo). A *flag* –O3, além de habilitar as mesmas otimizações da *flag* –O2, habilita também as otimizações: *prefetching, scalar replacement e loop transformation*. A *flag* –ip habilita otimizações entre procedimentos (*interprocedural optimizations*) tais como: *inline function expansion, interprocedural constant propagation*, etc. Uma das configurações que utilizamos em nossos experimentos (referenciada como Intel–O2.IP) para o compilador Intel utiliza as *flags* de otimização –O2 e –ip. Uma outra configuração empregada (referenciada como Intel–O3.IP) utiliza as *flags* de otimização –O3 e –ip. Em ambas as configurações foi utilizado *profiling* com entradas iguais às utilizadas em cada programa do SPECint95.

O compilador SGI foi alterado para considerar uma quantidade maior de unidades funcionais (15 unidades funcionais para cada tipo) daquela existente na arquitetura Itanium™ e também para considerar latência 1 para as instruções. A *flag* –O2 habilita a maioria das otimizações do compilador. As otimizações neste nível são conservadoras, no sentido de que são sempre benéficas, provendo melhorias proporcionais ao tempo gasto na compilação. Já a *flag* –O3 habilita otimizações mais agressivas (nem sempre benéficas) que as habilitadas pela *flag* –O2. Uma das configurações do compilador SGI que utilizamos para nossos experimentos (referenciada como SGI–O2) utiliza a *flag* de otimização –O2. Uma outra configuração (referenciada como SGI–O3) utiliza a *flag* de otimização –O3.

Para estabelecer um valor máximo de desempenho no ambiente Trimaran, o item de configuração *unlimited resources* foi habilitado. Esta configuração foi combinada com os parâmetros para formação de blocos (superblocos e hiperblocos) do Trimaran e especificam máquinas com recursos infinitos para cada forma de escalonamento. Estas duas configurações são referenciadas neste trabalho como configurações Trimaran MAX SB e MAX HB, respectivamente. Uma outra configuração do ambiente Trimaran define um processador com 15 unidades funcionais de cada tipo existente (*integer*, *floating point*, *memory* e *branch*), 128 registradores de cada tipo existente (gpr, fpr, pr e btr), e latência 1 para todas as instruções. Esta configuração também foi combinada com os parâmetros de formação de blocos (superblocos e hiperblocos) do Trimaran. Estas configurações são referenciadas neste trabalho como configurações Trimaran 15 SB e 15 HB, respectivamente. Foi utilizado *profiling* com entradas iguais às utilizadas em cada programa do SPECint95 em todas as configurações Trimaran. Estas configurações são bastante otimistas do ponto de vista de implementação.

As *flags* –O3 e –unrolloops foram utilizadas para o compilador gcc para gerar o código para a ISA Alpha.

Utilizamos os parâmetros *default* para o simulador SKI.

Para o simulador DTSVLIW os parâmetros utilizados estão mostrados na Tabela 2. O número de instruções executado e indicado nos resultados inclui apenas as instruções que seriam executadas em uma máquina escalar.

Os *caches* de dados e instruções de todas as máquinas em estudo foram consideradas ideais (sem penalidade de *miss*).

Todos os parâmetros utilizados foram escolhidos de modo a isolar o que está em estudo: a qualidade do escalonamento dos conjuntos compilador/ISA/arquitetura de máquina.

TABELA 2
PARÂMETROS DA 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 15x16 instruções, 3072-Kbyte
Cache de Instruções	Perfeito (sem penalidade de *miss*)
Cache de Dados	Perfeito (sem penalidade de *miss*)
Número de registradores para *renaming*.	128

B. Experimentos e Resultados

B.1 Compilador Intel x SGI x Trimaran

A Figura 3 mostra o número de instruções de cada programa do SPECInt95 executadas pelos simuladores EPIC em estudo. Os programas foram compilados pelos compiladores EPIC Intel, SGI e Trimaran com as diferentes configurações descritas na Subseção IV.A (Intel–O2.IP, Intel–O3.IP, SGI–O2, SGI–O3, Trimaran 15 SB, Trimaran 15 HB, Trimaran MAX SB, Trimaran MAX HB). A média aritmética (M.A.) do número de instruções executadas também é mostrada.

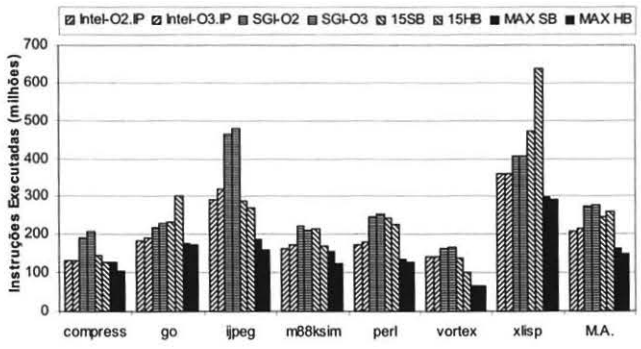

Fig. 3 EPIC: Instruções executadas

Como pode ser observado no gráfico da Figura 3, os compiladores Intel e SGI produzem pouca variação na quantidade de instruções executadas em função do nível de otimização utilizado. No entanto, na maior parte dos casos, o número de instruções executadas é maior quando a otimização é mais agressiva. Nas configurações Trimaran não podemos, na verdade, falar de otimizações mais ou menos agressivas, mas diferentes. Nas configurações com recursos limitados, 15SB e 15HB, *hyperblock scheduling* gera programas que executam menos instruções na maioria dos casos, embora esta tendência não seja confirmada nos casos de *go* e *xlisp*, possivelmente devido ao uso de *predication* em trechos que não são bons candidatos ao uso desta técnica. Já nas configurações Trimaran com recursos ilimitados, MAX SB e MAX HB, observamos um comportamento mais uniforme, onde a formação de hiperblocos leva sempre a um número menor ou igual de instruções executadas. Comparando os compiladores entre si, podemos ver pela Figura 3 que o compilador Intel sempre gera código que executa menos instruções que o compilador SGI e que o compilador Trimaran produz, na média, resultado entre Intel e SGI com recursos limitados ou executa menos instruções que estes dois com recursos ilimitados.

A Figura 4 mostra a número de ciclos consumidos na execução dos programas do SPECInt95 com as mesmas configurações citadas para a Figura 3. A média aritmética (M.A.) do número de ciclos consumidos também é mostrada.

O número de ciclos consumidos representa a melhor medida de desempenho, já que indica o tempo de execução de cada programa. Observando o gráfico, podemos perceber que a configuração Intel–O2.IP obteve, na média, desempenho ligeiramente superior que a configuração Intel –O3.IP (cerca de 2,3%). Isto mostra que nem sempre otimizações mais agressivas resultam em ganhos de desempenho. As configurações Intel, na média, obtiveram melhor desempenho que as configurações SGI (cerca de 19 %). A configuração SGI –O3, na média, obteve desempenho ligeiramente melhor que a configuração SGI –O2 (cerca de 3,3% melhor). As configurações Trimaran com recursos restritos, 15 SB e 15 HB obtiveram, na média, o pior desempenho. O número de instruções executados a mais para os programas *go* e *xlisp* nestas configurações tiveram um forte impacto negativo no desempenho. Como poderia ser esperado, as configurações Trimaran MAX SB e MAX HB obtiveram melhor desempenho em todos os programas – na média, cerca de 37,7% melhor que Intel –O2.IP, o segundo colocado. No entanto, diferente do que poderia ser esperado, a configuração com *hyperblock scheduling* gerou programas mais lentos que a configuração com *superblock scheduling* em todos os programas exceto *m88ksim* e *xlisp*, embora não substancialmente mais lentos.

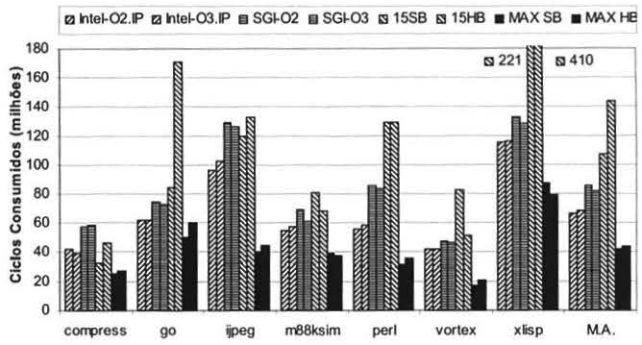

Fig. 4 EPIC: Ciclos consumidos

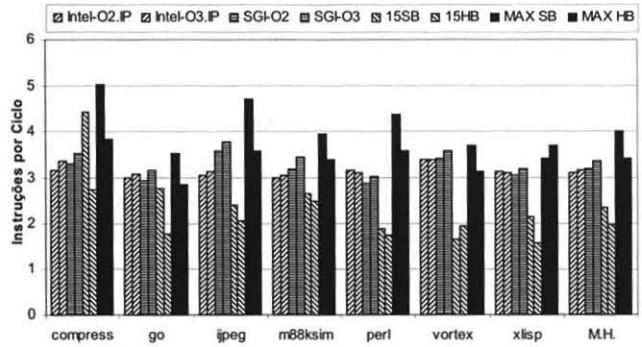

Fig. 5 EPIC: Instruções por Ciclo

A Figura 5 mostra o número de instruções por ciclo obtido na execução dos programas do SPECInt95 com as mesmas configurações da Figura 3. A média harmônica (M.H.) do número de instruções por ciclo também é mostrada. O número

de instruções por ciclo indica o grau de paralelismo no nível de instrução alcançado.

Como pode ser visto pelo gráfico da Figura 5, os compiladores Intel e SGI não conseguem paralelismo no nível de instrução muito maior que três – o tamanho do *bundle* da ISA IA64. Note ainda que, muitas vezes, o compilador preenche *slots* dos *bundles* com NOP's, que são contados como instruções executadas, o que também contribui para que o número de instruções por ciclo fique acima de três. As configurações Trimaran 15 SB e 15 HB obtiveram pior resultado em todos os programas exceto *compress* com a configuração 15SB. No entanto, mais uma vez, as configurações MAX SB e MAX HB obtiveram os melhores resultados na média, com vantagem para a MAX SB.

B.2 Desempenho DTSVLIW x EPIC

Para comparar o escalonamento EPIC com o escalonamento DTSVLIW, selecionamos a configuração de melhor desempenho de cada compilador EPIC estudado – Intel-O2.IP, SGI-O3 e MAX SB. A Figura 6 mostra o número de instruções executadas para cada uma destas configurações, que é igual aos valores já mostrados, juntamente com o número de instruções executadas pela DTSVLIW (na verdade, apenas as instruções que seriam executadas em uma máquina Alpha escalar). Como podemos observar na Figura 6, o compilador gcc produz programas que sempre executam menos instruções que os gerados pelos compiladores Intel e SGI. Isto é esperado. A capacidade de execução condicional da ISA Alpha é representada apenas por algumas instruções MOVE condicionais pouco usadas pelo compilador, enquanto que a ISA IA64, para isto, possui amplos mecanismos intensivamente usados pelos compiladores que levam à execução de muitas instruções extras. Compiladores para a ISA IA64 também geram muitas instruções NOP devido às restrições impostas pelo campo *template* para *bundles* válidos. Programas gerados pelo gcc executam, na média, a mesma quantidade de instruções que os gerados pelo Trimaran com recursos ilimitados, no entanto. O Trimaran não gera NOP's; assim, pelo menos com recursos ilimitados, nossos resultados mostram que é possível fazer um bom uso de *predication* no que diz respeito ao número executado de instruções.

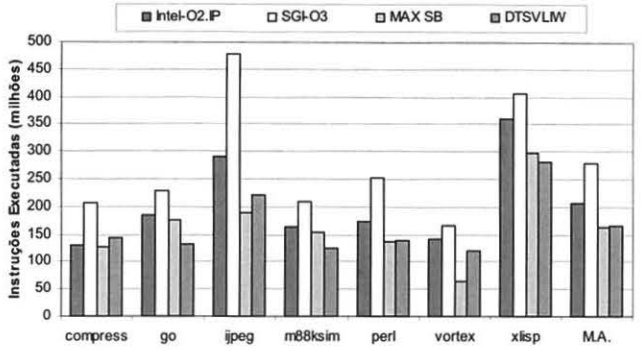

Fig. 6 DTSVLIWxEPIC: Instruções executadas

A Figura 7 mostra o número de ciclos consumidos durante a execução de cada um dos programas do SPECInt95. Como pode ser visto no gráfico da Figura 7, a combinação gcc/DTSVLIW supera as combinações compilador Intel/IA64 e compilador SGI/IA64 por uma larga margem na média. A DTSVLIW só perde em um programa, *go*, para a combinação Intel/IA64, mas por uma pequena margem. Comparada com a combinação Trimaran/HPL-PD, no entanto, gcc/DTSVLIW só ganha em *m88ksim* e *xlisp*; contudo, na média seu desempenho é equivalente.

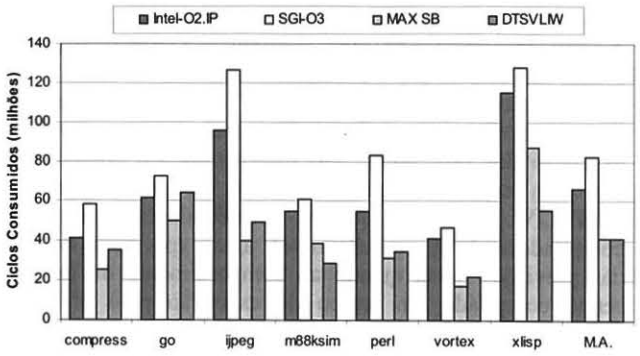

Fig. 7 DTSVLIWxEPIC: Ciclos consumidos

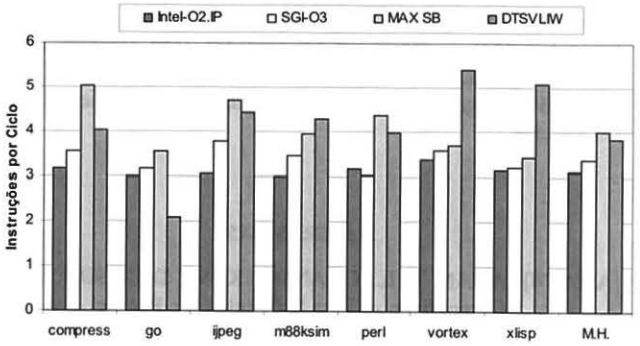

Fig. 8 DTSVLIWxEPIC: Instruções por ciclo

A Figura 8 mostra o número de instruções por ciclo obtido com os programas do SPECint95 para cada configuração em estudo. Como o gráfico da Figura 8 mostra, a combinação gcc/DTSVLIW conseguiu explorar níveis de paralelismo apenas inferiores que aos de um compilador EPIC compilando para uma máquina ilimitada e com código sendo executado em uma máquina com recursos ilimitados. A configuração da máquina DTSVLIW é bastante otimista, mas muito menos otimista que a máquina EPIC emulada pelo simulador Ski e ainda muito menos otimista que a emulada pelo simulador Trimaran. Em nossos resultados a DTSVLIW gastou, na média, praticamente o mesmo número de ciclos, obteve ILP 4,4% menor e executou 1,5% mais instruções que o Trimaran MAX SB. Mas comparado com a configuração Intel–O2.IP, que obteve resultado melhor que SGI –O3, a DTSVLIW, na média, gastou 37,6% menos ciclos, atingiu um ILP 23,4% maior executando 19,5% menos instruções.

V. Conclusão

Neste trabalho apresentamos uma análise comparativa experimental entre o escalonamento estático EPIC e o dinâmico DTSVLIW. Em nossos experimentos, o escalonamento DTSVLIW obteve, na média, praticamente o mesmo desempenho do obtido com o compilador EPIC Trimaran configurado com recursos ilimitados, o qual obteve o melhor resultado dentre os compiladores EPIC utilizados. Contudo, a máquina DTSVLIW utilizada obteve este desempenho utilizando uma configuração significativamente menos otimista do que a das máquinas EPIC.

Acreditamos que a DTSVLIW obteve o desempenho demonstrado porque informações dinâmicas a respeito da execução dos programas, não disponíveis aos compiladores, são bem exploradas pelo algoritmo de escalonamento da DTSVLIW. Enquanto que os escalonadores estáticos dos compiladores EPIC trabalham sob uma expectativa de acerto das previsões feitas quanto ao comportamento do programa sendo compilado em tempo de execução para obter bom desempenho.

A arquitetura DTSVLIW pode ter sido desfavorecida porque o código executado pelo simulador DTSVLIW foi gerado por um compilador não específico para esta arquitetura, enquanto que, para a arquitetura EPIC foram utilizados compiladores específicos. Além disso, as diferenças existentes entre as ISA's EPIC e a ISA Alpha não foram consideradas. Estas questões poderão ser esclarecidas com análises futuras de desempenho da DTSVLIW rodando código EPIC e/ou rodando código gerado por compilador específico para a DTSVLIW.

Referências

[ADA 74] ADAM T. L.; CHANDY, K. M.; DICKSON, J. R. A Comparison of List Schedules for Parallel Processing Systems. Communications of ACM, v.17, p.685-690, dec 1974.

[AHO 86] AHO, A.; SETHI, R.; ULLMAN, J. D.. Compilers - Principles Techniques and Tools. Addison-Wesley Publishing Company, USA, 1986.

[ALL 95] ALLAN, V. H. et al. Software Pipeline. ACM Computing Surveys, Vol. 27, No. 3, September 1995.

[DES 98] DE SOUZA, Alberto F.; ROUNCE, Peter. Dynamically Trace Scheduled VLIW Architectures. Proc. of the HPCN'98, in Lecture Notes on Computer Science, Vol. 1401, pp. 993-995, 1998.

[DES 00] DE SOUZA, Alberto F.; ROUNCE, Peter. Dynamically Scheduling VLIW Instructions. Journal of Parallel and Distributed Computing, n.60, p.1480-1511, 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 81] FISHER Josep. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, v. C-30, n.7, p.478-490, Jul 1981.

[FIS 84] FISHER, Josep. The VLIW Machine: A multiprocessor for Compiling Scientific Code. IEEE Computer, Jul 1984.

[GWE 97] GWENNAP, L. Intel, HP make EPIC Disclosure. Microprocessor Report, Vol. 11, No. 14, Oct 1997.

[HPL 98] HP Laboratories; New York University, ReaCT-ILP Group; University of Illinois, IMPACT Group. Trimaran: An Infrastructure for Compiler Research in Instruction Level Parallelism. 1998.

[HPL 00] HP Laboratories. Ski IA64 Simulator Reference Manual. Ver 1.0L, Apr 2000.

[HWU 93] HWU, Wen-mei W.; et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. The Journal of Supercomputing, v.7, p.229-248, 1993.

[INT 99] INTEL Corporation. IA64 Application Developer's Architecture Guide., 1999.

[INT 01] INTEL Corporation. Intel C++ Compiler User's Guide., 2001.

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

[KAT 94] KATHAIL, Vinod; SCHLANSKER, Michael; RAU, B. Ramakrishna. HPL-PD Architecture Specification: Version 1.0. HPL-93-80, Feb 1994.

[MAH 92] MAHLKE, Scott; LIN, David; CHEN, William; HANK, Richard; BRINGMANN, Roger. Effective Compiler Support for Predicated Execution Using the Hyperblock. In: Proceedings of the 25th Annual International Symposium on Microarchitecture, p.45-54, 1992.

[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, p.13-25, 1997.

[PAR 91] PARK, J. R. H.; SCHLANSKER, M. S. On Predicated Execution. Technical Report HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

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

[RAM 77] RAMAMOORTHY, C. C.; Li, H. F. Pipeline Architecture. ACM Computing Surveys, Vol. 9, No. 1, pp. 61-102, March 1977.

[RAU 93] RAU, B. R. FISHER, J. A. Instruction-Level Parallelism: History, Overview, and Perspective. The Journal of Supercomputing, Vol. 7, pp. 9-50, 1993.