Determinação dos Parâmetros Ideais de uma Arquitetura VLIW

Alberto Ferreira de Souza[1]
Edil Severiano Tavares Fernandes[2]

Resumo

Máquinas VLIW (*Very Long Instruction Word machines*) são arquiteturas paralelas que oferecem uma alternativa ao uso de Processadores Vetoriais e Multiprocessadores. Incorporando diversas unidades funcionais que podem operar em paralelo, essas máquinas podem ser caracterizadas pelas suas instruções, que incluem campos distintos para controlar diretamente cada um dos recursos do *hardware* subjacente. Esse trabalho descreve os experimentos que permitiram avaliar o efeito de importantes parâmetros arquiteturais no volume do paralelismo de baixo nível que pode ser extraido de programas de aplicação. Através da interpretação do código objeto de um processador comercial, derivado de uma bateria de programas de teste, determinou-se a configuração ideal da máquina VLIW capaz de atingir a taxa de aceleração máxima de cada programa de teste.

Abstract

Very Long Instruction Word (VLIW) machines are highly parallel architectures that can be used as an alternative to Multiprocessors and Vector Processors. These machines incorporate a large number of functional units that can be activated in parallel and can be characterized by their instructions, which include separated fields exercising direct control to the resources of underlying hardware. This work describes the experiments that have been carried out to assess the impact of important architectural parameters on the volume of low level parallelism that can be extracted from the application programs. By interpreting the actual object code derived from a suite of test programs, we have determined the ideal VLIW machine configuration leading to the maximum speedup that can be achieved by each test program.

[1]**Alberto Ferreira de Souza** é Engenheiro Eletrônico pela UFRJ, M.Sc. em Engenharia de Sistemas e Computação pela COPPE/UFRJ e professor do Departamento de Informática da UFES desde 1993. É, atualmente, Coordenador do Curso de Engenharia de Computação da UFES. Suas áreas de interesse são: arquitetura de computadores, processamento paralelo e redes neurais. D. I., Centro Tecnológico, UFES - Campus de Goiabeiras, Caixa Postal: 019011 - FCAA, 29060-970 - Vitória - ES, (027)-335.2654, alberto@inf.ufes.br asouza@npd1.ufes.br

[2]**Edil Severiano Tavares Fernandes** é Bacharel em Matemática pela UFRJ, M.Sc. em Engenharia de Sistemas e Computação pela COPPE/UFRJ e Ph.D. em *Computer Science* pelo Imperial College (Inglaterra). Atualmente é professor da COPPE-SISTEMAS/UFRJ. Suas áreas de interesse incluem arquitetura de computadores, sistemas operacionais e processamento paralelo. COPPE-SISTEMAS, C. T., UFRJ, Cidade Universitária, Rio de Janeiro - RJ, edil@cos.ufrj.br

1 Introdução

As máquinas VLIW (*Very Long Instruction Word*) [4] formam uma classe de processadores caracterizados por possuir:

- um único fluxo de controle (um contador de programa e uma unidade de controle);
- uma palavra de instrução longa (*long instruction word*), contendo bits para controlar, direta e independentemente (durante cada ciclo de processador), cada unidade funcional da máquina;
- um grande número de caminhos de dados e de unidades funcionais, cujo controle é determinado em tempo de compilação (isto é, cuja ativação é determinada previamente). Máquinas VLIW não têm árbitros de barramento, filas ou outros mecanismos de sincronização por *hardware*.

O uso de unidades funcionais *pipelined* e a ausência de *hardware* para gerência de conflitos, tornam as máquinas VLIW simples e rápidas. Uma unidade funcional nunca precisa esperar pelo resultado de outra, o que permite que o *hardware* opere sempre na sua velocidade máxima.

Neste trabalho são apresentados experimentos que foram realizados para determinar quais as dimensões que uma máquina VLIW deve ter, para executar uma classe de programas de aplicação com máximo desempenho. Para isso foram desenvolvidas três ferramentas que, juntas, formam um ambiente para determinar os parâmetros ideais de uma máquina VLIW.

A primeira destas ferramentas é um simulador do processador i860. Através desse simulador é possível gerar um *trace* da execução de um conjunto de programas de teste (*benchmark*). A segunda ferramenta é um programa capaz de produzir, a partir do código objeto de cada programa de teste, uma descrição detalhada de cada instrução presente neste código objeto. A terceira ferramenta é um compactador de código, que produz um código paralelo mapeado em uma máquina VLIW (cuja configuração pode ser determinada através de parâmetros) a partir do *trace* e da descrição do código objeto de um programa de teste. As características da máquina VLIW (isto é, seus parâmetros) são passadas para o compactador sob a forma de um arquivo.

Através da compactação de cada um dos componentes da bateria de testes para máquinas VLIW com diferentes características, foi possível obter o perfil da VLIW que melhor se adequa a cada programa de teste.

2 Máquinas VLIW

As máquinas VLIW (*Very Long Instruction Word*) são arquiteturas altamente paralelas que oferecem uma alternativa aos Processadores Vetoriais e Multiprocessadores [4]. Elas se assemelham a máquinas microprogramadas horizontais e são caracterizadas por:

- Uma unidade de controle central que inicia uma única instrução longa (*long instruction*) por ciclo;
- Um grande número de *data paths* e unidades funcionais, sendo o controle destas planejado em tempo de compilação;
- Cada instrução longa inicia simultaneamente diversas operações nas diversas unidades funcionais que compõem a máquina;
- Cada operação requer um pequeno e estaticamente previsível número de ciclos para ser executada;
- Não existem árbitros de barramento, filas ou outros mecanismos de sincronização por *hardware* na unidade de controle;
- Algumas operações são *pipelined.*

As VLIW não se enquadram na classificação de Flynn [1] - não são MIMD nem SIMD. Por permitirem a execução de diferentes instruções em um mesmo ciclo não podem ser classificadas como SIMD e, por seu fluxo de instruções ser centralizado, não podem ser classificadas também como MIMD. Apesar disso, do mesmo modo que os Processadores Vetoriais (SIMD) e os Processadores Paralelos (MIMD), as VLIW reduzem o tempo total de execução dos programas através da exploração de seu paralelismo, sendo que nestas isto é feito de forma muito fina - diversas instruções são empacotadas em uma mesma *long instruction.*

Diferente dos Processadores Vetoriais, as VLIW não requerem uma grande regularidade no código para fazer uso efetivo da potencialidade do seu *hardware.* E diferente dos Multiprocessadores, não há nelas perdas por sincronização ou comunicação. Todas as unidades funcionais executam seu codigo completamente sincronizadas e diretamente controladas, em cada ciclo do relógio, pelo compilador.

Processadores Vetoriais e Processadores Paralelos não possuem compiladores que possam produzir bom código paralelo para um grande número de programas seqüenciais. Já as máquinas VLIW, por se assemelharem muito a máquinas microprogramáveis horizontais, são, como estas, quase impossíveis de programar manualmente de forma eficiente. A tarefa de produzir código para elas deve ficar a cargo do compilador. Usando técnicas como *Trace Scheduling* [2,6] ou *Percolation Scheduling* [7], um compilador para VLIW liberta o programador da difícil tarefa de localizar o paralelismo e explorá-lo dentro da

estrutura do *hardware*. Assim, as VLIW podem explorar fortemente o paralelismo de granularidade fina e, com as técnicas citadas de compilação e otimização de código, o paralelismo de granularidade grossa existente nos programas [3]. Isso torna o uso de máquinas VLIW uma maneira prática de se obter *speedup* para a grande maioria das aplicações computacionais.

Experimentos demonstraram que o paralelismo disponível a máquinas VLIW é muito grande, ficando, em muitos casos, limitado apenas pelo tamanho dos dados [10]. Mas qual deveria ser o "tamanho" das máquinas VLIW para atender com bom desempenho aos programas mais típicos? É certo que não se deve projetá-las para casos particulares, com um grande número de unidades funcionais, já que elas seriam subutilizadas em situações onde não houvesse tanto paralelismo disponível. Por outro lado, projetando-as com poucas unidades, poderíamos estar perdendo a oportunidade de aumentar o *speedup* em muitos casos típicos. Assim, realizar medidas para se saber qual é a configuração mais apropriada para as máquinas VLIW se torna necessário.

A seguir, é apresentada uma descrição do ambiente desenvolvido para determinar a configuração, isto é, número de unidades funcionais, barramentos, canais de comunicação com a memória, tamanho do banco de registradores, etc, que uma máquina VLIW deve ter para casos típicos.

3 O Ambiente de Avaliação

Dado um programa, qual deve ser a configuração de uma máquina VLIW para explorar seu paralelismo latente ao máximo? Esta foi a pergunta que se buscou responder com este trabalho através de experimentos. Não se desejava que a limitação imposta pelos desvios condicionais interferisse nas medidas - existe muito paralelismo disponível além dos blocos básicos (um bloco básico de código não tem desvios levando a ele, exceto no início, e não tem desvios deixando-o, exceto no final). Mas como evitar esta interferência? A solução foi supor que a máquina VLIW era capaz de prever o endereço resultante de um desvio condicional. Isto, embora seja um suposição muito forte, é verossímil, já que a máquina com esta capacidade seria equivalente a uma DataFlow com recursos ilimitados de *hardware* [11].

Para realizar os experimentos os seguintes estágios foram executados:

- escolha de um nível de linguagem onde o paralelismo pudesse ser encontrado;
- escolha de programas de teste dentro dos quais o paralelismo latente pudesse ser detectado e explorado pela máquina sendo avaliada;
- desenvolvimento de uma ferramenta que permitisse detectar o paralelismo dos programas e, para as diferentes configurações da máquina em estudo, avaliar o volume desse paralelismo.

3.1 O *Assembly* do Microprocessador i860

O nível de linguagem escolhido para detectar paralelismo foi o *Assembly*. No nível de *Assembly* é possível, através de procedimentos relativamente simples, fazer análises do código, o que em níveis mais altos se torna bastante complicado. A linguagem *Assembly* escolhida foi a do microprocessador Intel i860.

O microprocessador Intel i860 define uma arquitetura completa contendo unidades que realizam operações com inteiros, com reais e operações gráficas [12,13]. As operações do i860 são especificadas numa única palavra. Isso, somado ao fato das instruções serem RISC, permite identificar claramente que recursos do processador estão sendo utilizados em cada ciclo de máquina. Por estas características o *Assembly* do microprocessador i860 foi escolhido como nível de linguagem onde o paralelismo dos programas de teste seria detectado.

3.2 Os Programas de Teste

As máquinas VLIW foram consideradas, no passado, mais apropriadas para executar código científico. Dadas as suas dimensões e características, só aplicações científicas pareciam justificar sua construção. Com o avanço da tecnologia de fabricação de circuitos integrados, que permite hoje a fabricação de dispositivos com centenas de milhares de gates, as máquinas VLIW já podem ser construídas para uso geral. Assim, os programas de teste escolhidos para realização dos experimentos foram programas típicos de máquinas de uso geral. São eles:

Livermore Loop 24 - algoritmo para determinação do menor componente de um vetor de dimensão n [14];

Quick Sort - algoritmo de ordenação;

Busca Binária - algoritmo de busca;

Bubble Sort - algoritmo de ordenação, método da bolha;

Decomposição LU - método para solução de equações lineares baseado na Eliminação Gaussiana;

Integração numérica - algoritmo de integração numérica segundo o método do trapézio.

Estes programas foram escritos em linguagem C e posteriormente traduzidos para o *Assembly* do i860 pelo compilador HighC, da Metaware [15].

3.3 As Ferramentas de Avaliação

Para obter as medidas de desempenho deste trabalho era preciso uma ferramenta que permitisse prever precisamente a direção tomada por cada desvio condicional, já que numa máquina VLIW é essencial explorar o paralelismo além dos blocos básicos. Por outro lado, a inclusão de um mecanismo de previsão dinâmica de desvios numa arquitetura VLIW é uma decisão de projeto que contraria radicalmente um dos princípios que caracterizam processadores desse tipo (isto é, a complexidade do *hardware* deve ser minimizada de modo a aumentar a velocidade de processamento da máquina resultante) e por essa razão, o algoritmo de escalonamento das instruções que serão executadas simultaneamente durante cada ciclo da máquina VLIW é implementado numa fase que precede a interpretação do código objeto. Esse algoritmo de escalonamento usualmente é implementado (através de *software*) pelo compilador, durante a fase de otimização de código. Deste modo, decidiu-se produzir um *trace* dos programas da bateria de testes e em seguida, através de um algoritmo de compactação de código, foi medido o paralelismo existente em cada um dos componentes da bateria de testes.

O *trace* é uma ferramenta de avaliação muito poderosa, pois permite visualizar globalmente a execução do programa, exatamente o que é necessário para realizar as medidas almejadas. No *trace*, todos os desvios têm o seu destino determinado, e por esse motivo, o programa todo é transformado em um único bloco básico!

Para se obter o *trace* dos programas de teste foi desenvolvido um simulador do processador i860, o Sim860 [5]. Os programas de teste foram processados pelo compilador de C e posteriormente simulados no Sim860. Durante a execução, o Sim860 gera um arquivo contendo o *trace* do programa que está sendo simulado.

Além do Sim860 foi contruída outra ferramenta para realização dos experimentos: um gerador de código estático. Este programa, denominado Scode [5], gera um arquivo contendo uma descrição detalhada de cada instrução utilizada no programa, a partir do executável criado pelo compilador. De posse deste arquivo e do arquivo de *trace*, é possível alimentar o programa na máquina VLIW sendo avaliada. Aqui, alimentar o programa na VLIW sendo avaliada significa escalonar cada operação presente no *trace*, de acordo com os recursos existentes na máquina VLIW, e obedecendo as restrições impostas pelas dependências de dados existente no programa, ou seja, compactar o código. Como este trabalho foi feito utilizando-se um *trace*, a tarefa de compactação se resume na compactação local (compactação dentro de blocos básicos apenas), já que um *trace*, do ponto de vista da compactação, se comporta como um único bloco básico.

Para realizar a tarefa de compactação de código foi desenvolvida uma terceira ferramenta: um compactador de código. Esse compactador, denominado Compact [5], gera um código paralelo mapeado no *hardware* da máquina VLIW em estudo, a partir do arquivo de *trace* e do arquivo de código estático. Ele gera também, estatísticas para a avaliação dos resultados obtidos com cada configuração de máquina VLIW em estudo.

3.4 Técnica de Compactação Escolhida

Muitos trabalhos foram feitos, por diversos pesquisadores, sobre compactação local e global de código [16,17,18,19]. Vários algoritmos de compactação local foram profundamente avaliados por David Landskov e Scott Davidson nos trabalhos [8,9]. Através dessas avaliações, foi escolhido o algoritmo FCFS (*First-Come First-Served*), pelo seu bom desempenho e por ser apropriado ao problema.

A seguir, o conceito de compactação de código é formalmente apresentado, assim como o algoritmo FCFS. Apresenta-se, também, uma descrição do modelo de máquina VLIW em estudo.

4 Compactação de Código

A expressão "Compactação de Código" foi inicialmente utilizada para especificar uma tarefa relacionada à microprogramação de máquinas microprogramáveis horizontais.

Um microprograma é uma seqüência de microinstruções. Eles são armazenados em uma memória especial, chamada memória de controle, e as microinstruções que os compõem são executadas uma de cada vez (normalmente uma a cada ciclo de máquina). Durante a execução, as microinstruções são as palavras de controle da máquina. Cada atividade do processador, especificada dentro de uma microinstrução, é chamada de microoperação. Máquinas microprogamáveis horizontais são caracterizadas por possuirem microinstruções largas, nas quais diversas microoperações podem ser alocadas. Compactar o código é escolher, dentre os diversos arranjos possíveis de microoperações, aquele que minimize o tamanho do microprograma e, conseqüentemente, seu tempo de execução.

Neste trabalho, compactação tem significado equivalente. Onde se tinha microinstruções, temos Instruções Longas (IL) e onde se tinha microoperações, temos Instruções (IN). O problema de compactação pode então ser definido como:

> **Compactação** - Para uma determinada máquina VLIW é dado um programa definido por uma seqüência de INs. Compactar este programa consiste em alocar estas INs nas ILs, de modo que o tempo de execução do programa seja minimizado. Essa alocação deve respeitar duas restrições:
>
> 1. A máquina deve possuir recursos (unidades funcionais) disponíveis para executar as INs alocadas em cada IL.
> 2. A seqüência resultante de ILs deve ser semanticamente equivalente à seqüência original de INs.
>
> Por "semanticamente equivalente" deve-se entender que, se ambas as seqüências são executadas com a mesma entrada, a saída deve ser sempre a mesma.

Para se garantir que não haja alteração semântica no programa, devem ser observadas as "dependências de dados" entre as INs durante a compactação.

4.1 Dependência de Dados

A maioria das INs de uma máquina opera sobre registradores. Um registrador cujo valor é usado por uma IN é chamado de registrador de entrada ou operando de entrada. Analogamente, um registrador cujo valor é alterado por uma IN é chamado de registrador de saída ou operando de saída. Uma IN pode ter também, como operandos de entrada ou saída, posições de memória ou flags.

Dado um programa para ser compactado, a lista final de ILs obtida após a compactação deve ser semanticamente equivalente ao programa original. Para produzir um código eficiente, o processo de compactação modifica a ordem original das INs, movimentando-as ao longo da lista de ILs. Como conseqüência dessa movimentação, a ordem de execução das INs é alterada: o processo de compactação pode antecipar (ou retardar) o início da execução de algumas INs. Em algumas situações, o algoritmo de compactação precisa manter o seqüenciamento especificado pelo programador de modo a preservar a integridade do programa de aplicação. A ordem de execução de duas INs não pode ser alterada se elas interagirem. A interação de duas INs (denominada também Interação de Dados) pode ser definida como:

> **Interação de Dados** - Ocorre entre duas INs, IN_i e IN_j, onde IN_i ocorre antes que IN_j no programa original, satisfazendo as seguintes condições:
>
> 1. Um operando de saída de IN_i é usado como operando de entrada de IN_j.
> 2. Um operando de entrada de IN_i atua como operando de saída de IN_j.
> 3. Um operando de saída de IN_i também é um operando de saída de IN_j.

Se IN_j preceder IN_i no programa compactado, o seguinte pode ocorrer:

- Se a primeira condição for violada, IN_j pode utilizar um valor antigo como operando de entrada ao invés do novo valor gerado por IN_i.
- Se a segunda condição for violada, IN_j pode destruir um valor antes que IN_i o tenha utilizado.
- Caso a terceira condição seja violada, o valor final de um operando (após a execução de IN_i e IN_j), que deveria ser avaliado por IN_j, recebe o valor produzido por IN_i. Se este valor não for mais utilizado como operando no programa, esta condição pode ser desconsiderada.

A lista de ILs resultante da compactação será semanticamente equivalente à lista original de INs se, para cada par de INs na lista final de ILs que apresente interação de dados, a IN ocorrendo antes na lista original de INs, termine de usar cada operando causando interação de dados antes que a outra comece a utilizá-los. Isto leva diretamente às seguintes definições:

Ordem Preservada - Considere IN_i e IN_j, duas INs de um programa, com IN_i ocorrendo antes que IN_j. As duas INs têm ordem preservada se, para cada operando causando interação de dados entre elas, IN_i liberar esses operandos antes que IN_j comece a utilizá-los.

Dependência de Dados Direta - Considere IN_i e IN_j, duas INs de um programa, IN_i ocorrendo antes que IN_j. IN_j possui dependência de dados direta com IN_i, ou IN_i **ddd** IN_j, se as duas INs apresentarem interação de dados e se não existir uma seqüência de INs com dependência direta de dados entre elas. Isto é, não existir um seqüência de INs, IN_{m1}, IN_{m2}, ..., IN_{mn}, $n \geq 1$ tal que IN_i **ddd** IN_{m1}, IN_{m1} **ddd** IN_{m2}, ..., $IN_{m(n-1)}$ **ddd** IN_{mn}, IN_{mn} **ddd** IN_j.

Dependência de Dados - Considere novamente IN_i e IN_j, duas INs de um programa, IN_i ocorrendo antes que IN_j. IN_j apresenta dependência de dados com IN_i, ou IN_i **dd** IN_j, se IN_i **ddd** IN_j ou se existe uma IN, IN_k, tal que IN_i **ddd** IN_k e IN_k **dd** IN_j. Dependência de dados é o fechamento transitivo da dependência de dados direta.

Duas INs apresentando dependência de dados devem ter sua ordem preservada durante a compactação para que o programa compactado seja semanticamente equivalente ao programa original.

4.1.1 Dependência de Dados e *Pipeline*

Um tratamento especial deve ser dado àquelas INs que utilizam unidades funcionais *pipelined*. Uma IN que utiliza uma unidade funcioanal *pipelined* não pode ser considerada concluída enquanto seus operandos de saída não forem escritos. Assim, uma outra IN que tenha dependência de dados com esta, só pode ser alocada em uma IL que venha ser executada após o tempo de latência (em ciclos) da unidade funcional *pipelined* utilizada. Isto pode ser facilmente atingido na análise de dependência de dados utilizando-se instruções "*dummy*".

Uma IN que seria executada num *pipeline* com n estágios é substituída por n outras INs: uma com características idênticas (utilizando os mesmos recursos de entrada, saída, unidades funcionais, campos, etc...) não *pipelined* e $n-1$ INs *dummy*, com operandos de

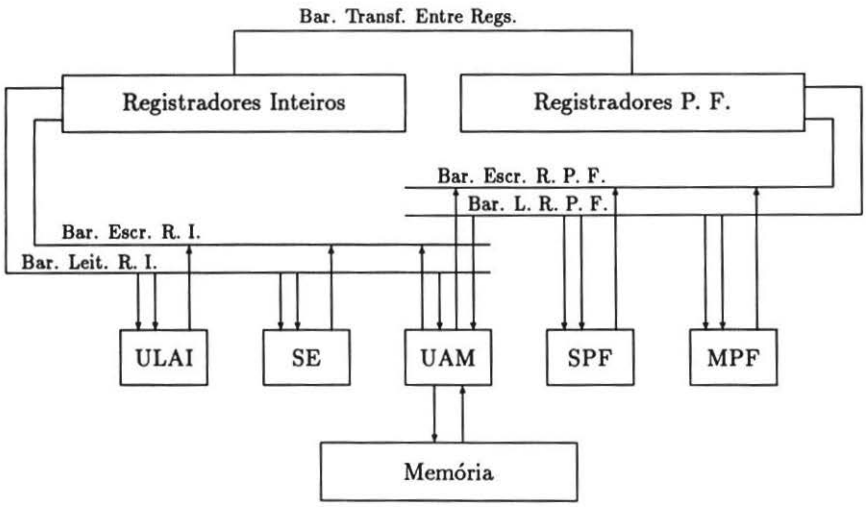

Figure 1: Máquina VLIW em Estudo

entrada iguais aos de saída e estes iguais aos de saída da IN que as originou. Assim, pelas regras de dependência apresentadas anteriormente, este conjunto de INs permanecerá unido na compactação, uma IN por cada IL de uma seqüência de ILs com tamanho n. Fica também garantido que INs com dependência de dados com uma IN *pipelined* não serão alocadas indevidamente, já que, também terão dependência de dados com as INs *dummy* [5].

4.2 Modelando a Máquina

Um modelo de máquina é necessário porque qualquer esquema de compactação deve produzir código levando em consideração as características da máquina onde ele será executado. Duas INs não podem ser alocadas na mesma IL se ambas necessitam de controle exclusivo de um mesmo recurso. Por exemplo: se existe uma única unidade lógica na máquina, apenas uma IN do tipo AND, ou qualquer outra operação lógica, pode ser alocada em cada IL.

O modelo de máquina usado nesse estudo pode ser visto na Figura 1 . Neste modelo a máquina é formada por componentes de 12 tipos:

1. U.L.A.I. - Unidade lógica e aritmética inteira;
2. S.E. - Somador de Endereços (responsável pelo cálculo dos desvios);

3. U.A.M. - Unidade de Acesso à Memória (responsável pelas leituras e escritas na memória);
4. S.P.F. - Somador de Ponto Flutuante;
5. M.P.F. - Multiplicador de Ponto Flutuante;
6. Banco de Registradores Inteiros;
7. Banco de Registradores de Ponto Flutuante;
8. Barramento de Leitura em Registradores Inteiros;
9. Barramento de Escrita em Registradores Inteiros;
10. Barramento de Leitura em Registradores de Ponto Flutuante;
11. Barramento de Escrita em Registradores de Ponto Flutuante;
12. Barramento de Transferência entre Registradores Inteiros e Registradores de Ponto Flutuante.

Esta máquina pode possuir mais de uma unidade funcional de cada tipo, um banco de registradores inteiros e um outro de registradores de ponto flutuante. Todas as unidades podem operar em paralelo. Uma IL para esta máquina tem campos para controlar cada uma das unidades, e cada campo deve ser preenchido por uma IN (ou um NOP). As INs podem ser de vários tipos e podem consumir mais que um campo, ou seja, utilizar mais que uma unidade funcional para sua execução.

4.3 Dependência de Recursos

Cada campo de uma IL está relacionado a uma unidade funcional da máquina VLIW. Os campos são especializados, embora possa existir mais de um campo com mesma finalidade, já que a máquina pode possuir réplicas de uma mesma unidade funcional.

Durante a compactação, ao se tentar alocar uma IN em uma IL, pode não haver campos disponíveis por já terem sido ocupados anteriormente por outras INs. Neste caso não é possível alocar a IN. Assim, uma IN não pode ser alocada em uma IL quando ela satisfaz à seguinte definição:

> **Dependência de Recursos** - Existe entre uma IN e uma IL quando não houver campo disponível em IL para alocar IN, por já ter sido, ou terem sido, ocupados por outras INs.

4.4 O Algoritmo de Compactação

O algoritmo de compactação utilizado foi o FCFS (*Fisrt-Come Fisrt-Served*), descrito inicialmente por Dasgupta e Tartar em [20]. Ele opera sobre a lista de INs que compõem o programa original, tomando uma IN de cada vez, na ordem em que elas aparecem no programa, e adicionando-as a uma lista (inicialmente vazia) de ILs.

A busca por uma IL, na qual a IN correntemente sendo examinada possa ser adicionada, inicia com uma análise de dependência de dados. Começando no fim da lista de ILs e seguindo para o início, vão sendo examinadas cada uma das ILs. A busca segue para uma IL mais próxima do início se a IN considerada não possuir dependência de dados com nenhuma IN na IL corrente. Se a IN tem dependência de dados com alguma outra IN na $i-ésima$ IL, a IN corrente não pode ser alocada em qualquer IL_j com $0 \leq j \leq i$. O objetivo desta busca é encontrar a IL de menor índice onde a IN possa ser alocada. Este índice é denominado *limite superior*.

O próximo passo para alocar a IN consiste em examinar os conflitos de recursos. Para ser alocada em uma IL, a IN corrente não pode apresentar conflitos de recursos com a mesma. Assumindo que o *limite superior*, i, foi encontrado, deve-se seguir em direção ao fim da lista de ILs, começando em IL_i, em busca de uma IL onde a IN corrente possa ser alocada. Quando esta IL for encontrada, ocorre uma inclusão. Se esta IL não for encontrada deve-se adicionar IN ao final da lista de ILs, através da criação de uma nova IL para receber IN.

Se não for encontrado um *limite superior*, a IN corrente não tem dependência de dados com nenhuma IN contida na lista de ILs e pode ser alocada em qualquer IL com a qual não tenha dependência de recursos. Neste caso, deve-se seguir do topo da lista em direção à ultima IL, em busca de um lugar para IN. Se esta busca falhar, cria-se uma nova IL no topo da lista para receber IN. Colocar esta IN no topo da lista irá evitar que ela bloqueie as INs subseqüentes que tenham dependência de dados com ela.

4.5 Renomeação de Registradores

Os compiladores, no processo de geração de código, podem associar variáveis a registradores. Diferentes variáveis podem ser associadas a um mesmo registrador; para isso, basta que estas variáveis estejam em diferentes procedimentos (ou subrotinas). Compiladores mais "espertos" podem até fazer a associação de mais de uma variável a um mesmo registrador dentro de um único procedimento.

Na compactação do *trace* estes casos gerarão falsas dependências de dados. Para contornar este problema foi utilizada a técnica de renomeação de registradores. Esta técnica consiste em trocar o nome de um registrador quando ele apresentar dependência de dados na compactação de uma IN, por satisfazer a 2ª ou a 3ª condição de interação de dados (vide Seção 4.1).

Ao banco de registradores inteiros e de ponto flutuante da máquina em estudo foram adicionados registradores extras para renomeação. Para otimizar o uso dos mesmos é feita análise de variáveis vivas (*live-variable analysis)* [21]. Nesta análise, uma variável pode estar viva ou morta em um determinado ponto do programa.

> **Variável Viva** - Uma variável está viva em algum ponto na execução do programa se ela vai ser lida em algum outro ponto, posterior a este, antes de uma nova escrita na mesma.

> **Variável Morta** - Se, em um determinado ponto do programa uma variável não está viva, então, ela está morta neste ponto.

No processo de compactação os registradores são marcados como vivos ou mortos. Registradores mortos podem ser utilizados para renomeação.

4.6 Os Parâmetros do Modelo

Ao longo dos experimentos para avaliar o efeito (no desempenho do modelo de VLIW) causado pela mistura dos diversos recursos do "*hardware*", várias configurações derivadas do modelo básico foram especificadas. Tendo em vista que o modelo básico é "parametrizado", cada conjunto distinto de parâmetros, resulta numa diferente configuração. Os parâmetros do modelo que foram investigados são as quantidades de: Unidade lógica e aritmética inteira; Somador de Endereços; Unidade de Acesso à Memória; Somador de Ponto Flutuante; Multiplicador de Ponto Flutuante; Barramento de Leitura em Registradores Inteiros; Barramento de Escrita em Registradores Inteiros; Barramento de Leitura em Registradores de Ponto Flutuante; Barramento de Escrita em Registradores de Ponto Flutuante; Barramento de Transferência entre Registradores Inteiros e Registradores de Ponto Flutuante; Registradores Inteiros para Renomeação; Registradores de Ponto Flutuante para Renomeação.

4.7 O Programa Compact

O programa Compact implementa o algoritmo de compactação FCFS e, durante sua execução, realiza diversas medidas. Este programa recebe como entrada três arquivos: um contendo um *trace* de um programa de teste (gerado pelo programa Sim860); um outro contendo o código estático deste programa (gerado pelo programa Scode); e um terceiro contendo o valor dos parâmetros especificando uma configuração da VLIW.

Durante sua execução, o programa Compact faz, na verdade, uma simulação da execução do programa de teste em uma configuração da máquina VLIW. Como saída, o programa Compact gera uma tabela como a Tabela 1 . Onde:

Resultado do Programa: LU.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
23.188	18110	26063	0.695	1124	16.112

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.677	0.112	0.059	0.149

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
10.915	18.065	11.023	3.348	2.399
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
0.880	0.920	4.109	3.318	0.059

Table 1: Informações Contidas no Arquivo de Saída do Programa Compact

Speedup - É a razão entre o número de ciclos requeridos pela execução do programa de teste em uma máquina seqüencial de referência e em uma máquina VLIW com código compactado. A máquina de referência é uma arquitetura constituída por uma unidade funcional de cada tipo. As latências de cada uma dessas unidades são iguais às das unidades do modelo de máquina VLIW. A máquina de referência executa uma instrução por vez;

Instruções - Número de instruções executadas na máquina seqüencial de referência;

Ciclos - Número de ciclos da máquina seqüencial de referência necessário à execução do programa não compactado;

I/C - Instruções / Ciclos;

Ciclos C. - Número de ciclos requeridos pela execução do programa compactado numa configuração de máquina VLIW;

I/C C. - Instruções / Ciclos C.;

Inteiras - Número de instruções lógicas e aritméticas inteiras;

Ponto F. - Número de instruções de soma, multiplicação, recíproco e comparação em ponto flutuante;

Desvio - Número de instruções de desvio (*jumps*, *jumps* condicionais, *calls*, etc);

Load/Store - Número de instruções de leitura e escrita na memória;

U.L.A.I. - Número de Unidades Lógicas e Aritméticas Inteiras;

B.L.R.I. - Número de Barramentos de Leitura em Registradores Inteiros;

B.E.R.I. - Número de Barramentos de Escrita em Registradores Inteiros;

Som. End. - Número de Somadores de Endereços;

U.A.M. - Número de Unidades de Acesso à Memória;

Som. P.F. - Número de Somadores de Ponto Flutuante;

Mul. P.F. - Número de Multiplicadores de Ponto Flutuante;

B.L.R.P.F. - Número de Barramentos de Leitura em Registradores de Ponto Flutuante;

B.E.R.P.F. - Número de Barramentos de Escrita em Registradores de Ponto Flutuante;

Bar. T. - Número de Barramentos de Transferência entre registradores inteiros e de ponto flutuante.

Executando-se diversas vezes o programa Compact tendo como entrada os arquivos de um mesmo programa de teste e diversos arquivos contendo parâmetros, pode-se traçar o perfil da máquina VLIW que melhor se ajusta à execução deste programa de teste.

5 Experimentos e Análise dos Resultados

Para a realização dos experimentos o seguinte procedimento foi seguido:

1. Tradução de um programa de teste com o compilador de C;
2. Geração de um arquivo de *trace* (através do programa Sim860) e de um arquivo de código estático (através do programa Scode), a partir do programa de teste compilado;
3. Execução do programa Compact utilizando os arquivos de *trace* e código estático gerados no passo anterior, e com arquivo de parâmetros indicando que a VLIW possui recursos ilimitados. Neste passo é determinado o *speedup* máximo que o programa de teste pode atingir;
4. Para cada parâmetro, execução do programa Compact várias vezes, variando o valor do parâmetro de um mínimo, até um ponto em que o *speedup* se iguale ao *speedup* máximo. Os demais parâmetros são considerados ilimitados;

Ao final deste procedimento, obtem-se a configuração da máquina VLIW que melhor se ajusta à execução do programa de teste.

-	LIVERMOR	QUICK	BINARIA	BOLHA	LU	INTEGRAL
ULAI	10	240	80	50	50	3
BLRI	16	240	135	100	100	6
BERI	12	240	80	60	50	3
SE	4	160	90	140	30	2
UAM	2	30	18	20	35	1
SPF					16	6
MPF					16	2
BLRPF					35	23
BERPF					28	8
BTER					1	3
RIPR	24	7000	570	4800	1100	3
RPFPR					1800	135
SPEEDUP	17.40	93.06	47.76	99.33	47.39	19.59

Table 2: Parâmetros Ideais para cada Programa de Teste

5.1 Análise dos Resultados

Na Tabela 2 são apresentados os parâmetros ideais da máquina VLIW obtidos com cada programa de teste. Pode-se constatar através dela que não existe uma máquina VLIW que possa ser utilizada de forma ótima por todos os programas de teste. Normalizando os parâmetros com relação ao número de ULAIs isto fica mais claro, como pode ser visto na Tabela 3. Mas, esta tabela também indica que existem relações entre os diversos parâmetros.

Existe uma relação entre o número de ULAIs, de BLRIs e de BERIs. Para cada ULAI de uma configuração, devem ser introduzidos 2 BLRIs, ou um número um pouco menor que 2 caso existam muitas ULAIs (mais que cinco ULAIs), e 1 BERI, ou um número um pouco maior, no mesmo caso. Estas relações existem por um motivo aparentemente óbvio: as ULAIs operam normalmente com dois operandos de entrada e um de saída. Como para algumas operações é necessário apenas um operando de entrada, o número de BLRIs tende a ser menor que 2 para cada ULAI. Por outro lado, outras unidades funcionais podem escrever nos registradores inteiros (os Somadores de Endereço, por exemplo), e por essa razão, o número de BERIs tende a ser maior que 1 para cada ULAI. Embora essas relações sejam aparentemente evidentes, elas poderiam ser diferentes. Outras unidades funcionais também fazem leituras nos registradores inteiros e isso poderia levar, por exemplo, a relação entre ULAIs e BLRIs para um número superior a 2. Contudo não é isso que mostraram os experimentos. Do mesmo modo, as ULAIs nem sempre escrevem nos registradores (no caso de operações de teste, por exemplo), e isso poderia levar a relação entre ULAIs e BERIs para um número inferior a 1. No entanto, os experimentos indicaram que este número é igual ou superior a 1.

-	LIVERMOR	QUICK	BINARIA	BOLHA	LU	INTEGRAL
ULAI	1	1	1	1	1	1
BLRI	1.60	1	1.69	2	2	2
BERI	1.20	1	1	1.20	1	1
SE	0.40	0.67	1.13	2.80	0.60	0.67
UAM	0.20	0.13	0.23	0.40	0.70	0.33
SPF					0.32	2.00
MPF					0.32	0.67
BLRPF					0.70	7.67
BERPF					0.56	2.67
BTER					0.02	1
RIPR	2.40	29.17	7.13	96.00	22.00	1
RPFPR					36.00	45.00
SPEEDUP	17.40	93.06	47.76	99.33	47.39	19.59

Table 3: Relação Entre o Valor de Cada Parâmetro e o Número de ULAIs

Essas relações são válidas também para as unidades de ponto flutuante, com a diferença de que existem dois tipos de unidades funcionais, específicas para ponto flutuante, que fazem leituras e escritas nos registradores de ponto flutuante. Conforme constatado pelos experimentos, o número de acessos (ao banco de registradores de ponto flutuante) é praticamente igual para os dois tipos de dispositivos funcionais (com uma ligeira predominância para os SPFs). Neste caso, normalizando pelo número de SPFs, para cada SPF devem ser incluídos oito BLRPF, ou um número um pouco menor, e dois BERPF ou um número também um pouco menor. É difícil determinar qual é a relação entre SPFs e MPFs com apenas dois programas de teste, mas, conforme mostrado pelos experimentos, deve-se ter um número entre 1 e 2 SPFs para cada MPF.

A relação entre ULAIs e SEs é uma das que mais varia de caso para caso. Foi possível observar, através de gráficos [5], que o parâmetro SE evolui de forma muito suave nos casos onde o número de SEs comparado com o número de ULAIs é muito grande. Através da observação dos programas compactados, verificou-se que isso ocorre porque em certos trechos dos programas podem ocorrer fortes concentrações de instruções que usam o mesmo tipo de unidade funcional, como é o caso das SEs. Assim, a redução do número de unidades funcionais desse tipo afeta apenas ligeiramente o *speedup*, afastando-o do ponto ótimo. Aceitando perdas de 5% a 10% no *speedup*, o número de SEs com relação ao de ULAIs poderia ser de 0.6 SEs para cada ULAI.

A relação entre o número de UAMs e o de ULAIs também varia bastante conforme o programa de teste e pelo mesmo motivo que a relação entre o número de SEs e o de ULAIs (embora não varie tanto quanto esta). Uma boa relação entre o número destas duas unidades funcionais seria de 0.3 UAM para cada ULAI, como indicaram os experimentos.

A relação entre o número de BTERs e o número ULAIs também varia muito nos dois experimentos em que os BTERs são utilizados, mas por um motivo completamente diferente. O conjunto de instruções do i860 não contém instruções para transferir constantes imediatas para os registradores de ponto flutuante. No i860 estas constantes devem ser montadas nos registradores inteiros e posteriormente transferidas para os registradores de ponto flutuante. No programa INTEGRAL, onde o número necessário de BTERs é grande (três) se comparado ao de ULAIs (também três), existem constantes que devem ser transferidas para registradores de ponto flutuante a cada passo do *loop* de INTEGRAL (o passo de integração e o limite superior da integral) o que gera um grande número de transferências entre registradores, conforme indicado pelos experimentos. No caso do programa LU, o número de transferências entre registradores é muito menor e, por conseguinte, a relação entre número de BTERs e o de ULAIs também.

O número absoluto de registradores para renomeação, de inteiros e de ponto flutuante, a princípio parece muito grande, mas, o seu número normalizado pelo número de ULAIs mostra que os totais obtidos nos experimentos estão dentro do que seria previsto. Este número normalizado nada mais é do que o número de registradores que devem ser incluídos para cada nova ULAI de uma VLIW. O número médio de registradores que devem ser introduzidos por ULAI está bem próximo do tamanho do banco de registradores dos microprocessadores de 32 bits (usualmente 32 registradores).

O que salta aos olhos nos resultados dos experimentos é a quantidade de paralelismo disponível nos programas. Um *speedup* de 99 é, sem dúvida, um *speedup* expressivo e pode ser conseguido por máquinas com arquitetura VLIW.

6 Conclusão

Este trabalho apresentou uma avaliação do impacto de importantes parâmetros arquiteturais no desempenho de Arquiteturas do tipo VLIW.

Interpretando o código objeto derivado de um conjunto de programas de teste, foram produzidos arquivos contendo *traces* da execução de cada um dos componentes da bateria de testes e, utilizando tradicionais técnicas de compactação de código, os arquivos de *trace* foram "paralelizados". Esse processo de compactação leva em conta as dependências entre as instruções do programa de teste e os recursos da máquina VLIW.

Para cada componente da bateria de testes, avaliou-se o impacto no desempenho provocado pela variação de cada um dos parâmetros individualmente (número de unidades funcionais, de barramentos, de registradores utilizados durante o processo de renomeação, etc). Através desses resultados, obteve-se a mistura ideal de componentes que deve constituir a arquitetura VLIW de modo a minimizar o tempo de execução de cada programa de teste.

Examinando os resultados dos experimentos, verifica-se que taxas de aceleração com

valores próximos de 100 foram obtidas por algumas configurações do modelo VLIW.

É importante observar que, apesar de extraordinário, esse desempenho das máquinas VLIW representa um limite superior da taxa de aceleração que pode ser atingida durante a execução de cada programa de teste: a utilização do arquivo de *trace* permitiu a remoção das incertezas provocadas pelos comandos de desvio condicional (o endereço da instrução sucessora de um comando de ramificação condicional sempre é conhecido durante o processo de compactação). Por esse motivo, foi possível transformar cada programa de teste num único bloco básico, viabilizando desse modo a extração de todo paralelismo existente.

A partir dos resultados obtidos nos experimentos, pode-se inferir que o processador cuja arquitetura originou nosso modelo de máquina VLIW (isto é, o processador i860 da Intel) ficaria mais balanceado se os seus projetistas tivessem optado pela inclusão de uma outra ULAI (Unidade Lógica e Aritmética Inteira) no *CORE* (dispositivo funcional que executa as leituras e escritas na memória, as operações inteiras e de controle do seqüenciamento). No "modo dual", esse processador é capaz de despachar duas instruções simultaneamente: uma para a unidade *CORE* e outra para a unidade de ponto flutuante. Esse modo dual de operação poderia ser utilizado mais eficientemente se duas instruções com inteiros pudessem também ser executadas em paralelo pelo *CORE* contendo uma segunda ULAI. Conforme indicado pelos experimentos, essa unidade adicional permaneceria em operação praticamente ao longo de todo o tempo de execução dos programas de nossa bateria de testes.

Referências Bibliográficas

[1] M. J. Flynn, "*Very High-Speed Computing Systems*", Proceedings of the IEEE 54, 1966, pp. 1901-1909.

[2] J. A. Fisher, "*Trace Scheduling: A Technique for Global Microcode Compaction*", IEEE Transactions on Computers, Vol. C30, No. 7, July 1981, pp. 478-490.

[3] D. D. Gagski and J. K. Peir, "*Essential Issues in Multiprocessor Systems*", Computer, Vol. 18, No. 6, June 1985, pp. 10-16.

[4] J. A. Fisher, "*The VLIW Machine: A Multiprocessor for Compiling Scientific Code*", Computer, July 1984, pp. 174-182.

[5] A. F. Souza, "*Avaliando os Parâmetros de uma Arquitetura VLIW*", Tese submetida ao corpo docente da COPPE/UFRJ, 1993.

[6] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman, "*A VLIW Architecture for a Trace Scheduling Compiler*", IEEE Transactions on Computers, Vol. 37, No. 8, August 1988, pp. 967-979.

[7] A. Nicolau, "*Percolation Scheduling: A Parallel Compilation Technique*", Technical Report, Department of Computer Science, Cornell University, May 1985, TR 85-678.

[8] D. Landskov, S. Davidson, and B. Shriver, "*Local Microcode Compaction Techniques*", Computer Surveys, Vol. 12, No. 3, September 1980, pp. 261-294.

[9] 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. C30, No. 7, July 1981, pp. 460-477.

[10] A. Nicolau and J. A. Fisher, "*Measuring the Parallelism Avaliable for Very Long Instruction Word Architectures*", IEEE Transactions on Computers, Vol. C33, No. 11, November 1984, pp. 968-976.

[11] Arvind and V. Kathail, "*A Multiple Processor Data Flow Machine that Supports Generalized Procedures*", Proceedings of the 8th Annual Symposium on Computer Architecture, ACM (SIGARCH), vol. 9, No. 3, May 1981, pp. 291-302.

[12] Intel, "*i860 64-Bit Microprocessor Hardware Manual - Preliminary*", Intel, Order Number: 240296-003, October 1989.

[13] Intel, "*i860 64-Bit Microprocessor Programmer's Reference Manual*", Intel, 1989.

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

[15] Metaware, "*HighC Programmer's Guide*", Metaware Incorporated, Santa Cruz CA, 1990.

[16] R. M. Tomasulo, "*An Efficient Algorithm for Exploiting Multiple Arithmetic Units*", IBM Journal, January 1967, pp. 25-33.

[17] M. Tokoro, E. Tamura and T. Takizuka, "*Optimisation of Microprograms*", IEEE Transactions on Computers, vol. C-30, No. 7, July 1981, pp. 491-504.

[18] M. Harris, "*Extending Microcode Compaction for Real Architectures*", ACM Micro, No. 20, 1987, pp. 40-53.

[19] S. U. Rao and A. K. Majumdar, "*Global Microcode Compaction - A Performance Evaluation by Simulation*", Microprocessing and Microprogramming, North - Holland, No. 20, 1988, pp. 159-174.

[20] S. Dasgupta and J. Tartar, "*The Indentification of Maximal Parallelism in Straight Line Microprograms*", IEEE Transaction on Computers, vol. C-25, pp. 986-991, October, 1976.

[21] A. V. Aho, R. Sethi and J. D. Ullman, "*Compilers. Principles, Techniques, and Tools*", Addison-Wesley, 1986.