PUC

Monografias em Ciência da Computação
nº 10/92

Geração e Otimização de Código: Levantamento dos Problemas e Restrições Impostas pelas Arquiteturas RISC e Indicativos de Soluções

Mariza Andrade da Silva Bigonha

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 10/92
Editor: Carlos J. P. Lucena Abril, 1992

Geração e Otimização de Código: Levantamento dos Problemas e Restrições Impostas pelas Arquiteturas RISC e Indicativos de Soluções

Mariza Andrade da Silva Bigonha

* Trabalho apresentado ao Prof. José Lucas Rangel.

Trabalho parcialmente financiado pela CAPES/UFMG e Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Para obter cópias:
Rosane T. L. Castilho
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22.453 - Rio de Janeiro, RJ.
Brasil

Abstract

This paper presents important problems related with RISC architectures that directly affect the generation of code generator systems and discusses the approaches proposed to solve them. Among these problems, we have the question about the interdependence between register allocation and instruction scheduling, the branch instruction, the restrictions imposed by pipelined processors and the phase ordering problems. This text concludes showing the disadvantages and advantages of RISC machine, and then compares the RISC and CISC architectures.

Keywords:RISC Architecture, Register Allocation, Instruction Scheduling, Code Generation, Optimization.

Resumo

Este texto contém uma investigação sobre os principais problemas provenientes das arquiteturas RISC que dificultam a tarefa dos geradores e otimizadores de código e algumas tendências da pesquisa corrente para obtenção de soluções para os mesmos. Dentre os problemas existentes ressaltam-se a interdependência entre a alocação de registradores e o escalonamento de instruções, as instruções de desvios, a presença de restrições impostas pelos processadores *pipelined* e o problema de ordenação entre fases. Apresenta também, algumas vantagens e desvantagens das arquiteturas RISC.

Palavras-chave: Arquitetura RISC, Alocação de Registradores, Escalonamento de Instruções, Geração de Código, Otimização.

Sumário

1 Principais Problemas Interligados às Arquiteturas RISC que Dificultam a Tarefa dos Geradores e Otimizadores de Código

1.1 Introdução

O objetivo deste texto é apresentar alguns dos mais relevantes problemas interligados às arquiteturas RISC (*Reduced Instruction Set Computers*), no que diz respeito à geração e otimização de código, e algumas soluções propostas para eles. Um destes problemas é a alocação de registradores versus o escalonamento de instruções. Neste contexto, é discutida a interdependência existente entre eles, apresentando as vantagens e desvantagens dependendo da escolha de qual deve ser efetuado primeiro.

Para as arquiteturas RISC são consideradas, também, as restrições impostas nessas máquinas pelos processadores *pipelined*, como, por exemplo: (a) se tais arquiteturas devem ou não possuir *pipeline interlocks*; (b) quais seriam as vantagens de tê-los por *hardware* ou por *software*; (c) os problemas e conseqüências para o gerador e otimizador de código; (d) e que soluções são encontradas hoje na literatura para a computação de uma seqüência de código na presença dessas restrições.

Instruções de desvios causam muitos problemas para máquinas e conseqüentemente para a geração de um código de qualidade. Desvios ocorrem com muita freqüência. Assim sendo, uma grande porcentagem do tempo de execução de programas é gasto desviando para diferentes instruções. Afim de minimizar a perda de ciclos de máquina durante sua execução, é importante saber como tratá-los. Este texto apresenta algumas abordagens para solucionar o problema dos desvios.

Outro aspecto importante, muitas vezes esquecido durante o projeto de uma nova arquitetura, diz respeito a irregularidade inerente das máquinas de um modo geral. Uma comparação baseada no código gerado pelo compilador para algumas arquiteturas de máquina mostra que a regularidade, também, é uma das peças fundamentais em um projeto de arquitetura de máquina.

Este trabalho conclui apresentando algumas vantagens e desvantagens das arquiteturas recentes.

Para abordar todos estes assuntos mencionados nos parágrafos anteriores, o trabalho foi dividido em três seções. O restante desta seção se encarrega da exposição dos principais problemas relacionados com essas arquiteturas, a interdependência entre alocação de registradores e seleção de instruções, os *pipeline interlocks*, as instruções de desvios e o problema

de ordenação entre fases na confecção de compiladores. A segunda seção apresenta as vantagens e desvantagens das arquiteturas RISC em relação às arquiteturas CISC. A conclusão mostra as considerações finais seguida da bibliografia utilizada na elaboração deste texto.

1.2 Interdependência entre a Alocação de Registradores e o Escalonamento de Instruções

O problema da interdependência entre o escalonamento de código e a alocação de registradores têm sido considerada uma das questões mais sérias para as arquiteturas paralelas e *pipelined* (RISC). Antes de expor este problema e algumas soluções propostas são introduzidos alguns conceitos relacionados com estas arquiteturas. O termo, processamento paralelo, por exemplo, refere-se a uma série de métodos, que, com o objetivo de acelerar o tempo de processamento de um programa, executa mais de uma computação concorrentemente. Um processador paralelo é um computador que implementa algumas técnicas de processamento paralelo, e como qualquer outro tipo de computação, o processamento paralelo pode ser visto sob vários níveis. No nível de circuitos, uma distinção é feita entre os circuitos de aritmética seqüencial e os circuitos de aritmética paralela. Naquela, ao processar um número examina-se 1 bit de cada vez. Nesta, todos os bits de um número são processados concorrentemente. No nível de registradores, onde a unidade de informação é a palavra, distingüem-se máquinas seqüenciais de paralelas baseando-se no fato de que uma ou mais palavras, instruções ou dados, podem ser processadas em paralelo [HAYES 88].

O termo *pipelining* é atribuído a uma técnica utilizada em computadores de alta velocidade. Esta técnica aprimora o desempenho do sistema sobrepondo a execução de instruções. Um processador *pipelined* é aquele no qual várias instruções seqüenciais executam simultâneamente, normalmente, em fases distintas. A presença de instruções de desvios, um outro problema dessas arquiteturas, e a dependência de dados existente entre algumas instruções, freqüentemente, restringem a eficácia de um longo *pipeline*. Bloqueios no *pipeline*, um mecanismo de *hardware* denominado *pipeline interlock* [HENNESSY 82, HENNESSY 83], são utilizados nestas arquiteturas para impedir, na presença de dependência de dados entre instruções, que a instrução posterior prossiga até que o valor requisitado da instrução anterior esteja disponível. Existem arquiteturas que não possuem bloqueios no *pipeline* por *hardware*. Em tais arquiteturas, uma alternativa é utilizar o escalonamento de instruções, técnica de *software* que rearranja seqüências de código durante a compilação com o objetivo de reduzir possíveis atrasos de execução.

1.2.1 Vantagens e Desvantagens do Escalonamento de Código *antes* da Alocação de Registradores

A técnica de escalonamento de instruções tem se mostrado efetiva na redução de *interlocks* no *pipeline*. Contudo, ela cria problemas para o alocador de registradores, quer pela sua execução anterior ou posterior à alocação de registradores. A vantagem de efetuá-la antes da alocação de registradores é que todo o paralelismo do programa pode ser explorado. A desvantagem é que ela aumenta o tempo entre a escrita em um registrador e sua posterior leitura, e pode provocar desperdício de alguns registradores. Essa desvantagem é a principal razão porque alguns pesquisadores preferem efetuar o escalonamento de instruções após a alocação. Por outro lado, o escalonamento de instruções após a alocação de registradores torna-se restrito porque o alocador de registradores pode, inadvertidamente, introduzir dependências atribuindo o mesmo registrador para instruções sem nenhuma relação. Na presença de pequenos blocos básicos esta restrição não afeta muito o desempenho do sistema, mas para blocos básicos maiores e *pipelines* longos esta restrição pode acarretar em uma grande perda no desempenho [GOODMAN 88]. A Figura 2 ilustra através de um exemplo, as vantagens e desvantagens do escalonamento de instruções *antes* e *após* a alocação de registradores para a seqüência de código que aparece na Figura 1.

f=a*b	1 Load R1,a	7	Load R7,e
g=(c+d)*(a+e)	2 Load R2,b	8	Add R8,R1,R7
h=f+g	3 Mul R3,R1,R2	9	Mul R9,R6,R8
	4 Load R4,c	10	Add R10,R3,R9
	5 Load R5,d	11	Stor R10,h
	6 Add R6,R4,R5		

Figura 1: Exemplo de um Bloco Básico

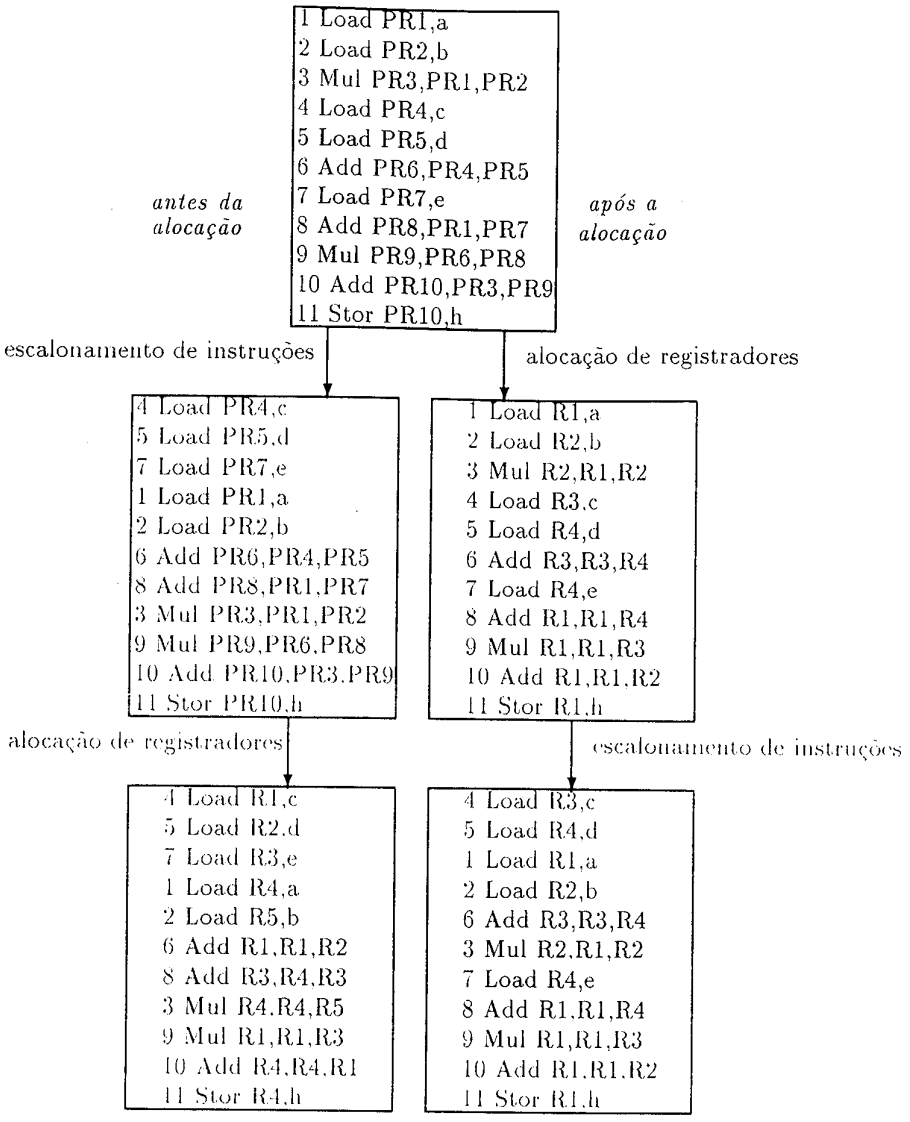

Figura 2: Escalonamento de instruções *antes* e *após* a alocação de registradores

Observe que a seqüência de código *antes da alocação* consome cinco registradores enquanto a seqüência de código *após a alocação* precisa de quatro registradores conforme mostrado na Figura 2. Se somente quatro registradores estão disponíveis à entrada do bloco básico, o código *antes da alocação* necessita carregar e armazenar instruções para liberar os registradores.

Existem técnicas que podem ser aplicadas durante a otimização de código para rear-

ranjar seqüências de instruções, entretanto, elas conflitam entre si. Exemplos destas técnicas: o escalonamento de código, denominado CSP (*Code Scheduling for Pipelined processors*) [HENNESSY 83, GIBBONS 86], utilizado para evitar possíveis atrasos de tempo de execução nas máquinas *pipelined*, e a reorganização de código, denominada CSR (*Code Scheduling to minimize Registers usage*) [DAVIDSON 86], usada para minimizar o número de registradores exigidos. CSP tende aumentar o tempo útil de cada pseudo-registrador, enquanto CSR quer diminuí-los. O escalonamento de código pode ser aplicado *antes* e/ou *após* a fase de alocação de registradores, enquanto somente faz sentido aplicar a técnica de reorganização de instruções *antes* da alocação de registradores.

Recentemente Davidson [DAVIDSON 86] separou a técnica de otimização, CSR, da fase de geração de código e a implementou como uma técnica de reorganização de código independente. A idéia principal desta otimização é evitar que um registrador mantenha um temporário ativo por um longo período de tempo.

Normalmente, como na Figura 2, uma técnica de alocação de registradores típica para um programa em uma linguagem intermediária requer quatro registradores. Contudo, utilizando-se o escalonamento de código para minimizar o uso de registradores (CSR) para a seqüência de instruções da Figura 1 foram necessários somente três registradores como ilustra a Figura 3. Nas Figuras 2, 3 e 4 PR seguido de um número representa um pseudo-registrador.

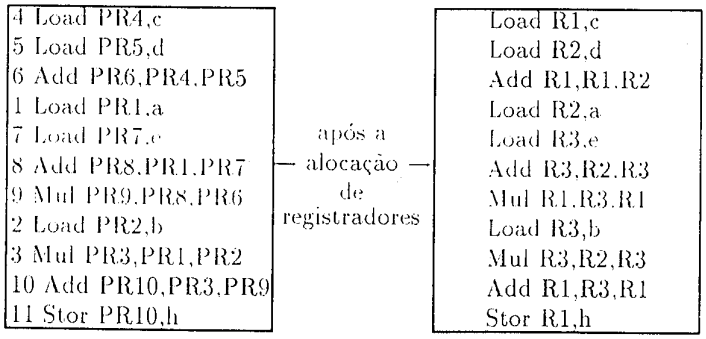

Figura 3: CSR minimizando o uso de registradores

Muito embora os problemas da alocação de registradores e do escalonamento de código sejam freqüentemente tratados em fases distintas no processo de compilação, há uma interdependência entre eles. James Goodman & Wei-Chung Hsu [GOODMAN 88] propoêm solucionar estes dois problemas, em uma única fase, durante a fase de otimização. Eles investigam duas abordagens: uma abordagem refere-se a alocação de registradores dirigida pelo escalonamento de instruções que combina as duas técnicas, CSP e CSR e a outra abor-

dagem refere-se a alocação de registradores dirigida por DAG (*Directed Acyclic Graph*).

1.2.2 Primeira Proposta de Solução

A proposta de solução de James R. Goodman & Wei-Chung Hsu [GOODMAN] para a primeira abordagem, alocação de registradores dirigida pelo escalonamento d nstrução, é integrar CSP e CSR em uma única fase para obter uma melhor seqüência g ódigo. A integração proposta ocorre com o escalonamento de instruções anterior ao a alocação de registradores com o objetivo de controlar o desperdício de registradores (*register spilling*). A idéia básica do método proposto é gerenciar o número de registradores disponíveis durante o escalonamento de código. Como cada instrução emitida pode requisitar novos registradores e finalizar com o uso de alguns outros é possível manter-se informado do número de registradores disponíveis. O escalonador utiliza CSP para reduzir os atrasos no *pipeline* quando há um número suficiente de registradores disponíveis. Na verdade, CSP é responsável pelo escalonamento de código durante a maior parte do tempo, somente quando o número de registradores diminui, caindo por exemplo para um, é que o escalonador desvia-se para o CSR com a incumbência de controlar o uso dos registradores. Após este número ser restaurado para um valor aceitável, CSP reassume o escalonamento. A troca entre CSP e CSR é dirigida, portanto, pelo número de registradores disponíveis. Inicialmente, este valor é determinado pelo número total de registradores da arquitetura em questão menos o número de registradores utilizados à entrada do bloco básico. Esta informação é obtida através da análise global de fluxo de dados [AHO 86]. Um contador de referências é utilizado para determinar pseudo-registradores que não são mais utilizados, liberando-os para uso. O número de registradores disponíveis é acrescido quando há registradores livres e decrementado quando instruções requisitam novos registradores.

Para ilustrar o funcionamento do método proposto, suponha que somente quatro regis- dores estejam disponíveis para o bloco básico apresentado na Figura 1. Após utilizar o escalonador de código de James Goodman & Wei-Chung Hsu obtém-se a seguinte seqüência de código:

```
4 Load PR4,c
5 Load PR5,d
7 Load PR7,e
1 Load PR1,a
```

Neste ponto o escalonador deve decidir entre emitir a instrução 2 que requisita um novo registrador e a instrução 6 que libera registradores. Como os registradores disponíveis foram todos utilizados, CSR é invocado e emite a instrução 6. Posteriormente o controle

retorna ao CSP e a instrução 2 é emitida após a instrução 6. A Figura 4 apresenta o resultado obtido para a seqüência completa do bloco básico da Figura 1 após o uso do escalonamento integrado.

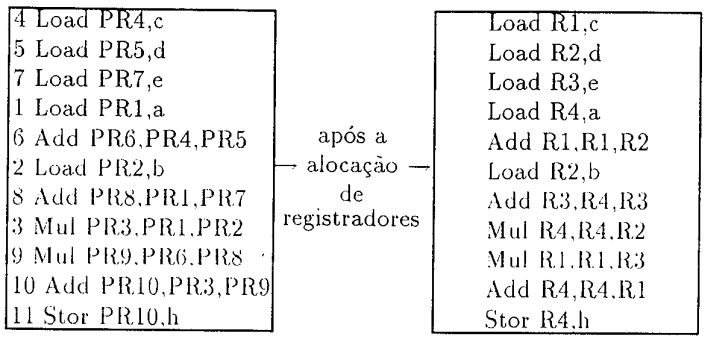

Figura 4: Seqüência de código utilizando o Escalonamento Integrado

Observe que esta seqüência utiliza o mesmo número de registradores da sequência exibida na Figura 2 para o escalonamento de instrução *após a alocação* de registradores. entretanto, a seqüência de código da Figura 4 possui menos bloqueios de tempo de execução.

1.2.3 Segunda Proposta de Solução

A proposta de solução de James R. Goodman & Wei-Chung Hsu para a segunda abordagem. alocação de registradores dirigida por DAG, consiste em um método de escalonamento para depois da alocação de registradores. Algoritmos de escalonamento, normalmente, reordenam as instruções para aprimorar o tempo de execução de programas. Esta ordem deve preservar parcialmente a ordem original imposta pelas restrições de precedência. Os DAGs (Grafos Acíclicos Dirigidos) são, geralmente, utilizados para representar as restrições de precedência de programas [AHO 86] definindo ordens de avaliações válidas dentro dos blocos básicos. Em um DAG os nodos representam instruções e as arestas indicam dependências entre instruções. Por exemplo, uma aresta partindo do nodo A para o nodo B indica que a instrução A deve ser executada antes da instrução B na seqüência de código. Nesta abordagem, um DAG de dependência é utilizado para auxiliar na atribuição de registradores.

Para introduzir o conceito de alocação de registradores dirigida por DAG são definidos mais dois termos: largura e altura de um DAG. A largura de um DAG é o número máximo de nodos mutuamente independentes que necessitam um registrador destino. Uma instrução de armazenamento (*store*) não necessita registrador destino. A altura de um DAG é o

comprimento de seu caminho mais longo. Caso o número de registradores reais de uma arquitetura seja maior que a largura do DAG, a forma do DAG pode permanecer inalterada durante toda a alocação de registradores. Por outro lado, se o número de registradores for menor que a largura do DAG, o alocador de registradores reduzirá a largura do DAG através do reuso de registradores para que ele fique menor ou igual ao número de registradores reais. Enquanto a largura é reduzida, a altura é acrescida, porque cada reuso de registrador pode fundir dois caminhos de avaliação em um. Quanto maior a altura, mais longo o caminho mais crítico, portanto, menos eficiente o escalonamento de código.

A segunda proposta de solução visa reduzir a altura do DAG reconstruído com o auxílio de duas estratégias: explorar novas dependências criadas pelo reuso de registradores, principalmente *write-after-read* (WAR) e equilibrar o crescimento do DAG. A presença de dependências causadas por *write-after-read* reduz o paralelismo disponível e resulta em um escalonamento de código menos eficiente. A dependência WAR é utilizada, essencialmente, para forçar a ordem lógica das instruções.

Para minimizar o crescimento em altura do DAG, o alocador de registradores procura em primeiro lugar selecionar um registrador de forma que somente dependências redundantes sejam introduzidas. Dependências redundantes são aquelas dependências introduzidas onde já existiam arestas de dependências entre os nodos impostas pela ordem lógica das instruções. Essas dependências são consideradas *dependências livres* porque não aumentam o tamanho do DAG.

Toda substituição de registradores que adiciona novas dependências pode aumentar a altura do DAG. Quando não há mais "dependências livres", os registradores são atribuídos baseados no "*earliest issue time*" (EIT) e no "*earliest finish time*" (EFT). O *earliest issue time* de um nodo é o custo máximo de um caminho a partir do início do DAG até o nodo em questão. O *earliest finish time* de um nodo é o custo máximo do caminho do nodo até o final do DAG.

A idéia central da segunda abordagem para minimizar a altura do DAG é equilibrar o crescimento dos mesmos. Para isto, o alocador de registrador tenta não conectar dois nodos, onde um nodo possui grande *earliest issue time* e o outro tem um grande *earliest finish time*. Caso a instrução corrente possua um alto EFT, então o alocador seleciona um registrador que não esteja sendo utilizado de forma que todos os nodos que a instrução corrente venha a conectar tenha um pequeno EIT.

1.2.4 Comparação Entre as Duas Propostas de Solução

Comparando as duas abordagens propostas por James Goodman & Wei-Chung Hsu concluise que para um modelo possuindo um longo *pipelining*, o método de escalonamento inte-

grado efetuado antes da alocação de registradores é melhor, e, que para um modelo possuindo um *pipeline* médio, a abordagem de alocação de registradores utilizando DAG é mais satisfatória. Este resultado é explicado da seguinte forma: para um modelo com um longo *pipelining* onde bloqueios poderiam ser relativamente caros, a liberação de registradores pode ser proveitosa porque o escalonamento integrado antes da alocação de registradores pode facilmente acomodar liberações vantajosas. Esta flexibilidade permite a técnica de escalonamento integrado obter um desempenho melhor do que a abordagem de alocação dirigida por DAG. Por outro lado, a vantagem da abordagem da alocação dirigida por DAG é que ela não aumenta o tamanho do código de forma alguma. Sendo assim, ela é propícia para arquiteturas que desencorajam o desperdício de código.

Ambas as abordagens se mostraram eficientes na solução do problema da interdependência entre a alocação de registradores e o escalonamento de instruções. Conceitualmente a abordagem dirigida por DAG é mais simples. Ela tenta minimizar as dependências relacionadas com o armazenamento alocando, cuidadosamente, um número limitado de registradores disponíveis para um bloco básico. A alocação de registradores dirigida pelo escalonamento de instruções é mais flexível e mais agressiva que a alocação dirigida por DAG. Por exemplo, nas situações em que longos bloqueios são encontrados durante o escalonamento de instruções a abordagem integrada força a liberação. Isto é particularmente interessante quando muitos valores de um *loop* externo são carregados através de registradores deixando um número insuficiente de registradores para que os *loops* internos tenham um escalonamento decente. O método de escalonamento antes da alocação de registradores apresentado força o *loop* interno a liberar registradores provendo um número suficiente para o escalonamento de código.

1.3 Problema de Ordenação entre Fases

A maior parte dos compiladores, para reduzir a complexidade e simplificar sua implementação, é organizado em um conjunto de passos ou fases. Cada fase efetua uma função no processo de compilação. A atribuição das funções e a ordem de aplicação destas fases, em um compilador otimizador, são consideradas partes críticas do projeto. É extremamente difícil projetar as fases responsáveis pela geração e otimização de código de forma a operarem isoladamente, simplificando suas implementações, ainda que sejam capazes de se interagirem para produzir um bom código. O problema de projetar e ordenar tais fases desta forma é denominado "problema de ordenação entre fases" (*phase ordering problem*). Um exemplo clássico do problema de ordenação entre fases é a interação das fases de alocação de registradores e a de seleção de instrução do gerador de código. Em muitos compiladores, a alocação de registradores é efetuada antes da seleção de instruções. Uma das razões desta ordem, naturalmente, é que, para escolher uma instrução ou instruções mais eficientes para operar sobre um determinado valor é necessário saber se o valor está localizado em um registrador ou memória. Infelizmente, a seleção de instruções pode afe-

tar o número de registradores disponíveis para alocação. Outro exemplo, visto nas seções anteriores, é a interação entre a alocação de registradores e o escalonamento de instruções.

Manuel E. Benitez & Jack W. Davidson descrevem em [BENITEZ 88] a implementação de um compilador/ligador com o objetivo de evitar os problemas de ordenação entre fases. Um aspecto importante deste projeto é o fato de que as fases de síntese do compilador e o sistema de ligação (*linker*) utilizam uma única representação intermediária. Isto resulta em alguns benefícios. Em primeiro lugar, permite que as fases de síntese do compilador sejam efetuadas em qualquer ordem e repetidamente eliminando, potencialmente, o problema de ordenação entre fases, comum em compiladores que utilizam diferentes representações para as fases de geração e otimização de código. Em segundo lugar, permite que a seleção de código seja feita em qualquer ponto durante as fases de síntese, bem como, em tempo de ligação. O fato de permitir a seleção de código em tempo de ligação apresenta várias oportunidades para otimizações que não podem ser efetuadas pelo compilador.

A representação intermediária utilizada em [BENITEZ 88] é descrita em forma de RTL (*register transfer list*). Uma RTL representa de forma independente de máquina uma operação dependente da arquitetura da máquina. Por exemplo, a RTL

$$r[1] = r[1] + r[2];\ cc=r[1] + r[2]?0;$$

representa uma soma de inteiros contidos em registradores em muitas arquiteturas. Todas as fases do compilador, inclusive a fase de ligação, podem manipular as RTLs. Existe uma série de razões para o uso de RTLs como base de representação intermediária: (1) devido sua forma ser independente de arquitetura, programas podem ser construídos para manipularem RTLs de formas independentes de máquina; (2) devido ao fato de as RTLs representarem instruções de máquinas, pontos específicos da arquitetura alvo são expostos às várias fases de otimização resultando em uma otimização mais completa; (3) finalmente, devido a boa formação das RTLs, é possível construir reconhecedores para determinar quando uma RTL representa uma instrução correta da máquina alvo. Esta habilidade de determinar, em qualquer ocasião, quando uma RTL representa uma instrução de uma determinada arquitetura constitui o ponto chave da estratégia de otimização de código implementada por Benitez e Davidson.

O compilador implementado por Benitez & Davidson é organizado em três partes distintas. A primeira parte, o *front-end* do compilador, denominado VPCC (*Very Portable C Compiler*), compreende os analisadores léxico, sintático e semântico e o gerador de código. No VPCC, estas três fases lógicas operam seqüencialmente. O VPCC emite um código intermediário orientado à pilha, denominado *C-code*, que é similar em espírito ao *P-code* [BERRY 78] exceto que ele modela máquinas RISC. *C-code* é composto de quarenta e três operações e, graças a generalidade do código é possível interpretá-lo como sugere

[DAVIDSON 87a]. A segunda parte, denominada CGEN, é composta do expansor de código (*code expander*) cujo papel é traduzir cada *C-code* para o código específico da arquitetura alvo no formato de RTLs. A terceira parte do compilador, VPO (*Very Portable Optimizer*) é responsável pela estratégia de geração e otimização de código. O VPO é composto de várias fases conforme ilustra a Figura 5.

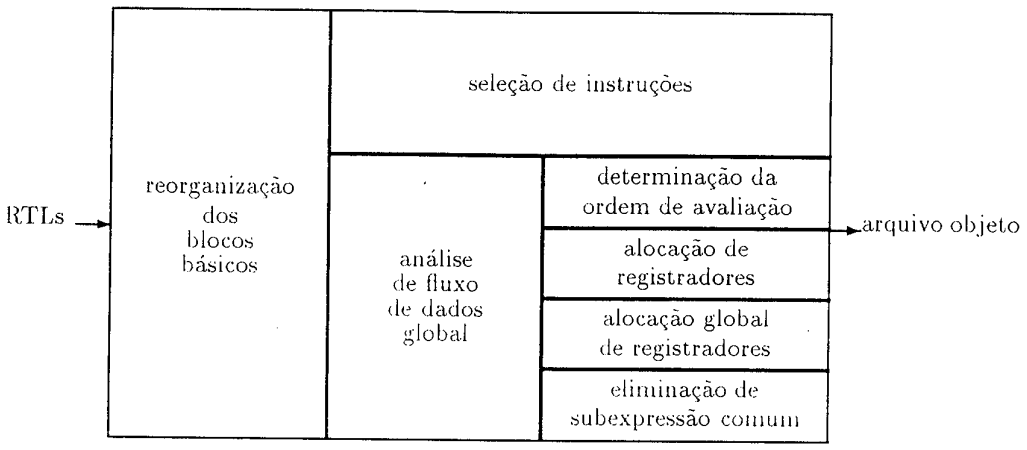

Figura 5: Esquema do Otimizador VPO utilizado no compilador C

As colunas verticais, na Figura 5, especificam as fases lógicas que operam seqüencialmente. As colunas divididas horizontalmente em linhas indicam subfases da coluna que podem ser executadas em qualquer ordem. Observando a Figura 5 nota-se que a fase de reorganização dos blocos básicos deve ocorrer antes de todas as outras fases. Esta otimização reduz o número de desvios. Neste ponto a seleção de instruções pode ser efetuada via otimização local [DAVIDSON 81] em cada bloco básico. Esta otimização, por sua vez, restringe o volume de RTLs que deve ser processada pela fase de análise de fluxo de dados global. Uma característica importante desta abordagem é a habilidade de efetuar a seleção de instruções em qualquer momento e repetidamente durante o processo de otimização, após, naturalmente, a fase de reorganização dos blocos básicos. As demais subfases podem ser ativadas em qualquer ordem. O passo seguinte do VPO é a análise de fluxo de dados global, que é efetuada logo no início com o objetivo de determinar informações referentes às variáveis mortas ou ativas necessárias à alocação de registradores. Após esta fase, a seleção de instruções é executada novamente para detectar as otimizações, apontadas pela análise de fluxo global, que podem ser efetuadas através dos blocos básicos. A determinação da ordem de avaliação e a alocação de registradores são efetuadas após a análise de fluxo de dados global e de todas as subseqüentes seleções de instruções necessárias, e seguida pela execução da alocação global de registradores. Neste ponto o alocador global de registradores sabe exatamente o número de registradores disponíveis para a atribuição. Utilizando

informações provindas do *front-end* e informações oriundas da análise de fluxo de dados globais, seleciona-se as variáveis que podem ser alocadas aos registradores. Custos também são utilizados para classificar as variáveis candidatas à alocação de registradores. Após a alocação, a seleção de instrução é novamente efetuada. A última subfase do VPO é o eliminador de subexpressão comum. Como o VPO opera no nível de máquina, subexpressões comuns existentes no código são detectadas e eliminadas [DAVIDSON 84b]. Se o eliminador de subexpressão comum fizer qualquer modificação no código, a seleção de instrução é reinvocada, e, se após a seleção de instrução, registradores ficarem disponíveis, alocador de registradores é também reativado.

O código objeto produzido pelo compilador é uma representação linear da estrutura de dados operada pelas fases de síntese do compilador. O sistema de ligação lê e processa este código executando as funções normais de um ligador. Adicionalmente ele efetua várias otimizações. Após ler o arquivo de entrada, o ligador constrói um grafo de chamada de procedimentos estendido (*extended call graph*) que mostra o fluxo de procedimento a procedimento. O ligador tenta efetuar otimizações através das chamadas de procedimentos e retornos. Uma otimização comum neste ponto é a remoção de instruções de testes desnecessários após a chamada de procedimento.

VPO opera no VAX-11 e SUN-3 e é composto de 5741 linhas de código C. Não mais que 15% de VPO necessita modificações para ser aplicado em uma nova arquitetura ou nova linguagem. O ligador possui 1184 linhas de código C e aproximadamente 30 linhas de código deve ser redirecionado manualmente.

1.4 Geração e Otimização de Código na Presença de Restrições Impostas pelos Processadores *Pipelined*

Outro problema tratado neste texto está relacionado com as restrições impostas pelos processadores *pipelined* nas arquiteturas RISCs [HENNESSY 82, HENNESSY 83], [GIBBONS 86] e [BERNSTEIN 89]. Enquanto nas arquiteturas mais tradicionais cada instrução é carregada, decodificada e executada antes que a próxima instrução seja carregada, nas arquiteturas *pipelined* os ciclos de execução de instruções distintas podem se sobrepor [GIBBONS 86]. Problemas surgem devido a dependência de dados existente entre duas instruções. Esta dependência aparece se o resultado de uma instrução é necessário para outra antes que a primeira tenha finalizado sua execução. Nesse caso, a segunda instrução deve esperar até que o valor de seus operandos, computados pela primeira instrução, estejam prontos. Outro problema surge se um recurso da máquina é requisitado, simultaneamente, por duas instruções. Para solucionar este problema, em geral, atribui-se prioridades às instruções e a que tiver a prioridade mais alta obtem o recurso primeiro.

Alguns processadores *pipelined* identificam estas situações no *hardware*, em outros uma NOP (*No Operation*) é explicitamente colocada entre as duas instruções. Em ambas as situações, o tempo de execução do programa pode aumentar. Mais sério ainda, a presença do mecanismo de *interlock* por *hardware* em arquiteturas *pipelined* requer verificadores para o mesmo. Este fato, além de tornar o projeto dessas arquiteturas complexo, impõe uma sobrecarga no tempo de execução de todas as instruções, inclusive aquelas que não são afetadas pelo *interlock*. A eliminação deste *hardware* permite a simplificação do projeto de arquiteturas. O problema a ser resolvido portanto se resume em encontrar uma ordem de avaliação das expressões que reduza o tempo de execução na presença destas restrições do *pipeline*.

Atualmente, o uso de *software* para detectar e remover *interlocks* é considerada a abordagem mais prática e efetiva. Existem outros aspectos que contribuem para o interesse por arquiteturas sem *interlocks*. Do ponto de vista do *software*, por exemplo, se forem providos algoritmos de reordenação de código, que permitam instruções não afetadas pelo *interlock* serem sempre executadas, sem que nenhum tempo seja gasto resolvendo os mesmos, o código executará mais rápido do que um código desordenado rodaria em uma arquitetura *pipelined* com *interlocks* por *hardware*. A única desvantagem aparece quando NOPs são inseridas nas seqüências de código, mesmo assim, há apenas um acréscimo de espaço. Do ponto de vista de *hardware*, a simplicidade e regularidade podem ser cruciais para que implementações de processadores sejam bem sucedidas.

1.4.1 Arquiteturas sem *Pipeline Interlocks* por *Hardware*

Os trabalhos de John Hennessy & Thomas Gross [HENNESSY 82, HENNESSY 83] investigam que conseqüência teria para o gerador de código o fato das arquiteturas *pipelined* não possuírem *pipeline interlocks* por *hardware*. Sua ausência por *hardware* faz com que seqüências de código geradas não executem corretamente. Esses *interlocks* devem, portanto, serem providos por *software* para evitar a execução de seqüências indefinidas. O compromisso do gerador de código, para tais arquiteturas, é garantir a execução correta do programa original, ou seja, assegurar que a função "entrada-saída" do programa seja idêntica quando o programa for executado de uma só vez, e quando for executado sem *interlocks* por *hardware* em arquiteturas *pipelined*.

Existem duas possíveis abordagens para solucionar este problema por *software*: (1) como um problema de escalonamento de instrução para ser resolvido durante a geração de código ou (2) como um problema de reorganização que deve ser resolvido após o código ter sido gerado.

Jonh Hennessy & Thomas Gross optaram pela reorganização de código para eliminar os *interlocks* existentes. A mais importante vantagem desta abordagem é que a reorganização

pode ser aplicada no código gerado pelo compilador e no código em linguagem de montagem produzido pelo programador.

1.4.2 Arquiteturas com Mecanismo de *Interlocks* por *Hardware*

Os trabalhos de David Bernstein & Izodor Gertner [BERNSTEIN 89] e Phillip Gibbons & Steven Muchnick [GIBBONS 86] consideram as restrições impostas pelo processador *pipelined* na computação de uma seqüência de código em arquiteturas com *interlocks* por *hardware*.

Para solucionar o problema, David Bernstein & Izodor Gertner [BERNSTEIN 89] consideram uma máquina *pipelined* que emite instruções a todo ciclo de máquina. Nas situações em que uma instrução utiliza o resultado da instrução precedente no *pipe*, ela deve ser atrasada para garantir que o programa vai computar um valor correto. Supõe-se que a emissão de tais instruções seja atrasada no máximo um ciclo de máquina. Para tal modelo de arquitetura, dado um número ilimitado de registradores e localizações de memória, o objetivo é encontrar o menor escalonamento para uma dada expressão. O algoritmo proposto para alcançar a meta almejada é uma modificação do algoritmo de Coffman-Graham [COFFMAN 72] que provê uma solução "ótima" para o problema de escalonamento de tarefas em dois processadores paralelos.

Phillip B. Gibbons & Steven S. Muchnick [GIBBONS 86] propõem um algoritmo para reordenar as instruções em uma fase posterior à geração de código durante a compilação. Seu algoritmo utiliza uma representação de DAG para escalonar heuristicamente as instruções dentro de blocos básicos.

1.4.3 Considerações sobre as Soluções Adotadas

A consideração mais importante sobre a natureza das arquiteturas em análise encontrada em [HENNESSY 83] e [BERNSTEIN 89] é que *interlock* existem somente entre instruções que são conectadas com uma aresta em uma representação utilizando DAGs.

Em [HENNESSY 83], às vezes, o atraso máximo das instruções devido às restrições do *pipeline* é de dois ciclos de máquina dificultando encontrar uma ordem de avaliação "ótima". Em [GIBBONS 86] o problema é ainda mais complicado, pois são considerados casos onde podem existir *interlocks* entre instruções sem relação alguma entre os dados. O algoritmo proposto em [GIBBONS 86] procura solucionar este problema heuristicamente.

O segundo ponto em questão é se existem restrições adicionais relacionados com os recur-

sos da máquina. Por exemplo, um número limitado de registradores torna o problema de encontrar uma ordem de avaliação "ótima" intratável, mesmo em arquiteturas seqüenciais. Na opinião de Bernstein & Gertner, os algoritmos de escalonamento devem ser incorporados aos compiladores antes que a atribuição de registradores seja efetuada, apesar de todos os problemas já citados neste texto. Ao contrário de Bernstein & Gertner, Hennessy [HENNESSY 83] e [GIBBONS 86] tratam o escalonamento de código após a atribuição de registradores. No caso de John Hennessy & Thomas Gross, é necessário porque a arquitetura MIPS (*Microprocessor Without Interlocked Pipe Stage*), para a qual eles propuseram uma solução, não possui *hardware interlocks*. Neste caso, as novas restrições que surgem em conseqüência do uso incontrolado do mesmo registrador em porções diferentes do código limita possíveis reordenações de instruções. Para resolver essas restrições adicionais dos registradores [GIBBONS 86] utiliza uma representação especial de DAG em um algoritmo heurístico, enquanto [HENNESSY 83] usa uma representação usual de DAG, resultando em um algoritmo muito mais complexo que o de [GIBBONS 86].

1.5 Os Problemas causados pelas Instruções de Desvios

A implementação de instruções de desvios é uma das tarefas mais difíceis e um dos mais importantes problemas que devem ser tratados nos computadores *pipelined*. Essas instruções tendem a interromper o fluxo natural das instruções no *pipeline*. Na sua ocorrência, o conteúdo de partes do *pipeline* é descartado e o *pipeline* tem que ser recarregado acarretando uma diminuição no tempo médio de execução das instruções.

A importância atribuída ao tratamento da degradação do desempenho do sistema, devido a presença de instruções de desvios, deu origem a várias abordagens com o objetivo de propor novas técnicas para reduzir o custo destas instruções e avaliar as propostas de solução já existentes.

Um esquema que ficou muito popular com o advento das arquiteturas RISC foi o *delayed branch*. Nesta técnica, os desvios são implementados permitindo que a instrução seguindo o desvio seja executada enquanto o desvio está sendo processado [DAVIDSON 90]. Existem pelo menos dois problemas relacionados com esta técnica, a necessidade do compilador ou montador (*assembler*) de encontrar uma instrução para ser colocada após a instrução de desvio e o custo da execução do próprio desvio. Apesar destes problemas, esta técnica é utilizada nas máquinas MIPS de Stanford [HENNESSY 83], HP Spectrum [COUTANT 86], IBM 801 [RADIN 82] e RISC de Berkley [PATTERSON 85].

Branch folding [DITZEL 87b], outra técnica que reduz o custo de execução de desvios foi implementada na arquitetura CRISP [DITZEL 87a]. Nesta técnica, as instruções são decodificadas e colocadas em um grande *cache* de instruções. Cada instrução neste *cache* contém o endereço da próxima instrução a ser executada. Os desvios condicionais são

tratados como tendo dois possíveis endereços para a próxima instrução. Para determinar que caminho tomar, efetua-se a verificação do *flag* do código de condição no *Program Status Word Register* e o *static prediction bit*. Este *bit* é utilizado como um indicativo para o *hardware* se o desvio deve ser transferido ou não. Se a comparação for posta a uma distância razoável do desvio condicional, então a instrução correta pode ser carregada sem atrasos no *pipeline*. Caso o caminho errado seja escolhido, parte do *pipeline* deve ser descartado. Os problemas relacionados com este esquema incluem a necessidade de um *hardware* mais complexo para implementá-lo e um grande *cache* de instruções, porque cada instrução decodificada possui 192 bits de comprimento.

Prefetch branch target [LEE 84] é uma outra abordagem para reduzir o custo da execução de desvios. Nesta técnica quando um desvio é identificado, um mecanismo especial calcula e busca antecipadamente (*prefetch*) o objeto do desvio. Assim, se o desvio for efetuado, o objeto é carregado imediatamente no estágio de decodificação de instrução do *pipe* sem atrasos adicionais para a carga da instrução. Este esquema também exige um *hardware* muito complicado.

Deborah S. Coutant, Carol L. Hammond & Jon W. Kelly [COUTANT 86], além de utilizar o *delayed branch*, propõem um mecanismo denominado anulação (*nullification*) para as arquiteturas RISC com o objetivo de tornar o mais eficiente possível, a execução dos desvios mais comuns. Para desvios condicionais, o esquema de anulação permite que uma instrução seja executada somente em duas situações, se o desvio para trás for efetivado e no desvio para frente, quando o mesmo não é feito. Estas duas situações foram escolhidas porque elas cobrem os casos mais comuns e permitem o uso de todos os ciclos disponíveis. O desvio para trás é o mais freqüente em *loops* e quase sempre é executado várias vezes, portanto, o esquema de anulação que permite a utilização do ciclo extra se o desvio acontece faz com que o ciclo seja usado mais freqüentemente. Estes mesmos argumentos são válidos para o desvio para frente. Como no exemplo ilustrado na Figura 6, desvio condicional que aparece no final de *loop* será, quase sempre, seguido pela cópia de primeira instrução do *loop*. Esta instrução será executada somente se o desvio acontecer.

```
L1    LDW          4(sp),r1        ; primeira instrução do loop
      .
      .
      .
      comibt,>=,N  10,r2,L1+4      ; desvio para L1+4 se 10 >= r2
      LDW          4(sp),r1        ; cópia da primeira instrução do loop
```

Figura 6: Desvio Condicional Para Trás

Um desvio para frente implementando um comando *if* será quase sempre seguido pela primeira instrução da cláusula *then*, permitindo o uso do ciclo sem o rearranjo do código. Esta instrução somente será executada se o desvio não acontecer conforme ilustra a Figura 7.

```
        comibf,=,N   0,r1,L1   ; desvie se r1 não for igual a 0
        addi         4,r2,r2   ; primeira instrução da cláusula then
        .
        .
        .
'L1:
```

Figura 7: Desvio Condicional Para Frente

Davidson & Whalley [DAVIDSON 90] descrevem uma técnica que elimina razoavelmente o custo de execução das instruções de desvios pelo uso de um novo conjunto de registradores. A grande maioria dos RISCs possuem trinta e dois registradores (MIPS, Spectrum, RS6000*) etc. Davidson & Whalley exploram a possibilidade de utilizar alguns destes registradores para conter os endereços das instruções de desvios e da instrução correspondente a cada objeto do desvio.

Neste esquema toda instrução especifica a localização da próxima instrução a ser executada. Para obter isto sem um grande aumento no tamanho da instrução, um campo dentro de cada instrução especifica um registrador que contém o endereço virtual da próxima instrução a ser executada. Para instruções especificando que a próxima instrução a ser executada é a próxima instrução seqüencialmente, o registrador de desvio é referido como aquele contendo o apropriado endereço. Este registrador é na verdade o contador de programa (PC). Se a próxima instrução a ser executada não é a próxima instrução seqüencial, então código é gerado para calcular e armazenar o endereço virtual daquela instrução em um registrador de desvio diferente, e referenciar aquele registrador de desvio na instrução corrente.

Para avaliar a eficácia deste esquema, foram projetadas e emuladas duas máquinas. Uma delas possuindo trinta e dois registradores de propósito geral utilizados para referências a dados, e outra com dezesseis registradores para dados e dezesseis registradores para desvios. Os resultados obtidos mostram que a utilização de registradores para desvios pode efetivamente reduzir o custo associado à transferência de controle.

2 Prós e Contras das Arquiteturas RISC

Há um consenso quase que geral entre os pesquisadores da área de linguagens, envolvidos em projeto de compiladores para arquiteturas RISC, de que tais arquiteturas no início possuíam uma série de vantagens sobre as arquiteturas CISC. Atualmente, estas vantagens não são tão nítidas devido às inúmeras características incluídas naquelas arquiteturas. Contudo, elas foram muito difundidas e estão presentes no mercado. Esta seção procura mostrar os prós e os contras destas arquiteturas desde sua concepção e os artifícios utilizados para sanar as deficiências existentes.

Linguagens de programação introduzem abstrações as quais permitem ao programador ignorar detalhes de uma implementação. O suporte para abstração não deve se concentrar somente em melhorar a eficiência de uma implementação mas sobretudo prover mecanismos contra violações das abstrações. Há indícios de que projetos recentes de arquiteturas, os quais clamam ser simples e poderosos, alcançam eficiência delegando aos compiladores a solução de todos os problemas complexos da geração e otimização de código. Mesmo a preocupação com a eficiência aparece centralizada, na maioria das vezes, apenas em alguns aspectos da máquina, como por exemplo, no número de ciclos gastos pelas instruções e no tamanho da memória [WIRTH 87].

As arquiteturas *CISC*, como o próprio nome indica, são compostas de modos de endereçamento e instruções complexas. A vantagem destas arquiteturas é que o tempo de busca (*fetch*) de uma instrução e o tamanho do código são relativamente pequenos. Entre as várias desvantagens existentes ressaltam-se aquelas que impõem sérias restrições à eficiência: (1) a necessidade de compiladores para reconhecer entre várias instruções complexas a possibilidade de gerar instruções reduzidas; (2) e a necessidade de vários ciclos para a execução de uma instrução. Como era de se esperar, reações contrárias às arquiteturas CISC apareceram e, como conseqüência, marcaram o retorno de implementações em um único nível através das arquiteturas *RISC*, que, sob vários aspectos, relembram os conjuntos de instruções de vinte e cinco anos atrás. Esperava-se, também, que estas arquiteturas reduzissem a complexidade dos compiladores, mas sob diferentes aspectos isto não tem ocorrido porque, muito embora, as arquiteturas RISC simplifiquem o processo de compilação, elas exigem mais dos compiladores em outras áreas.

Uma distinção é estabelecida entre duas características das arquiteturas *RISC*. Originalmente, *RISC* era uma arquitetura onde cada instrução gastava exatamente um ciclo para sua execução, e referências à memória era tratada assincronamente como um dispositivo externo [PATTERSON 85]. Mais tarde o uso de registradores, relativamente maiores, foi adicionado como característica essencial dos *RISCs*. Este acréscimo resultou efetivamente em um *store* em dois níveis e fez com que o compilador controlasse o local das variáveis. A esperança por compiladores simples desapareceu muito rapidamente, e de fato as implementações de linguagens de programação de alto nível têm que fiarem-se em compiladores

bem sofisticados.

Para [COUTANT 86], o processo de compilação para máquinas *RISC* é simplificado em alguns pontos devido as características inerentes destas arquiteturas. Por exemplo, a geração de código pode ser direta porque além do número limitado de instruções, não há a necessidade de escolhas complicadas entre instruções com efeitos similares. Nessas arquiteturas, toda instrução aritmética, lógica ou condicional é efetuada entre registradores e todo acesso à memória é feito explicitamente através de *loads* e *stores*, liberando o compilador da necessidade de escolher entre instruções com múltiplos modos de endereçamento. Todas as instruções são do mesmo tamanho e possuem um número limitado de formatos. Simplificar a tarefa do gerador de código implica na simplificação do otimizador. Esta simetria do conjunto de instrução torna, por exemplo, a tarefa da substituição ou remoção de uma ou mais instruções durante a otimização de código uma tarefa mais amena.

Nas arquiteturas *RISC* [COUTANT 86] otimização é indispensável para usufruir de todas as vantagens das características da arquitetura. Além disto, o fato de possuir um grande número de registradores impõe ao compilador a tarefa de utilizá-los eficientemente. Os RISCs também colocam algumas responsabilidades extras sobre os geradores de código na emissão de código para certas construções das linguagens de alto nível, como por exemplo, a movimentação de *bytes*, as operações decimais, as chamadas de procedimentos. Como o conjunto de instruções destas arquiteturas não possui instruções complexas para auxiliar na implementação destas construções, os geradores de código são forçados a utilizar combinações de instruções simples para obter o mesmo efeito. Entretanto, mesmo nas arquiteturas CISC, uma análise de casos complexos (*complex case analysis*) é, normalmente, necessário para usar corretamente as instruções complexas. A maneira encontrada por [COUTANT 86] para contornar este problema para a família HP foi através de *millicode*, uma implementação de instruções complexas utilizando um pacote de instruções simples do *hardware* na forma de subrotinas. *Millicode* tem a mesma utilidade que os tradicionais *microcode*, mas é comum a todas as máquinas da família, ao invés de ser específica para cada uma. Do ponto de vista da arquitetura, *millicode* é apenas uma coleção de subrotinas como qualquer outra. Uma instrução *millicode* é executada ativando a apropriada subrotina *millicode*. O custo de sua execução, portanto, aparece somente quando a instrução é utilizada.

Do ponto de vista de Jack Davidson, as arquiteturas RISC oferecem várias vantagens sobre as mais complexas arquiteturas. Elas são mais fáceis de implementar, elas simplificam a seleção de código, e elas suportam as linguagens de alto nível pelo menos tão bem quanto as arquiteturas CISC. Jack Davidson mostra em seu trabalho [DAVIDSON 87a] que os princípios das arquiteturas RISC podem ser aplicados no projeto de máquinas virtuais para uso em interpretadores com benefícios similares. O tamanho reduzido do conjunto de instruções das CVMs (*C Virtual Machine*) reduz substancialmente o esforço necessário para codificar um programa para construir uma CVM. Máquinas virtuais com instruções complexas necessitam a implementação de um grande conjunto de instruções possuindo

instruções complexas, conseqüentemente são mais difíceis de serem construídas. Outro argumento favorável às arquiteturas RISC é que seus compiladores são mais simples que os compiladores para arquiteturas CISCs. Enquanto questiona-se se este argumento é aplicável a máquinas reais, Davidson mostra que ele se aplica a máquinas abstratas utilizadas para produzir compiladores redirecionáveis [DAVIDSON 84].

2.1 Comparação entre Arquiteturas RISC e Arquiteturas CISC

Normalmente, comparações entre máquinas RISC e CISC são difíceis de serem estabelecidas porque tais arquiteturas diferem não somente na complexidade de seus conjuntos de instruções, mas em muitas outras características. Jack W. Davidson & Richard A. Vaughan [DAVIDSON 87b] apresentam em seu trabalho os resultados de uma série de experiências realizadas para isolar e determinar o efeito de um conjunto de instruções complexas sobre o desempenho da memória de *cache* e o tráfico do barramento. Para efetuar esta experiência eles utilizaram uma técnica denominada *architectural subsetting* [PATTERSON 85]. Nesta técnica, o conjunto de instruções da máquina virtual é criado selecionando-se instruções e modos de endereçamento de arquiteturas existentes. A máquina virtual resultante é um subconjunto da arquitetura base.

Nesta experiência foram criadas três máquinas virtuais, *MAXVAX*, *MIDVAX* e *MINVAX*. Para estas máquinas foram delineados três subconjuntos do conjunto de instruções do *VAX* com diferentes níveis de complexidade. Compiladores para a linguagem Y [HANSON 81] foram construídos para as três máquinas. Y é semelhante a linguagem C, ela suporta compilação separada, procedimentos recursivos, expressões envolvendo tipos escalares, arranjos de inteiros, caracteres e reais. Os *back-ends* dos compiladores *Y* foram construídos utilizando-se *PO*, um otimizador local redirecionável [DAVIDSON 81]. Para cada máquina, um conjunto de programas *benchmark* foi compilado e o tamanho dos programas resultantes foi comparado nestas máquinas.

MAXVAX inclui 16 modos de endereçamento e a maior parte das instruções podem utilizar qualquer um destes modos como fonte ou destino, além de permitir formatos de instruções de dois ou três endereços. *MIDVAX* contém oito modos de endereçamento. Seu conjunto de instruções permite instruções aritméticas e lógicas somente no formato de dois endereços. Não possui instrução do tipo incremento, decremento, clear, etc. O conjunto de instruções da máquina *MINVAX* é bem reduzido, ele contém somente quatro modos de endereçamento. Toda operação lógica e aritmética é efetuada em instruções com dois endereços via registrador a registrador e a memória é acessada via instruções *load* e *store*.

Utilizando-se de subconjuntos da mesma máquina para representar arquiteturas possuindo conjunto de instruções com vários níveis de complexidade, as possíveis diferenças existentes entre máquinas que afetariam o tamanho do programa são eliminados. Outras

características importantes relacionadas com estas três máquinas está no fato de que elas possuem o mesmo número de registradores e utilizam a mesma codificação para instruções e modos de endereçamento que possuem em comum.

Com todos os outros fatores constantes, Davidson & Vaughan obtiveram os seguintes resultados: um conjunto de instruções simples pode resultar em programas que necessitam duas vezes e meio mais o tempo de memória que os mesmos programas com um conjunto de instruções complexo. A avaliação do desempenho do *cache* apontou que para pequenos *caches* a complexidade do conjunto de instruções afeta seriamente o *miss ratio*. Felizmente este aspecto do desempenho do sistema é solucionado aumentando o tamanho do *cache*. Por último analisou-se o volume do tráfico no barramento para as três máquinas. Neste item, mesmo com *caches* grandes, por exemplo, maiores que `64K bytes`, a máquina que possui um conjunto de instruções mais simples gera duas vezes mais tráfico que uma máquina com um conjunto de instruções complexo. Isto acontece porque máquinas possuindo conjunto de instruções simples necessitam mais instruções para implementar uma dada função.

Uma outra comparação interessante foi realizada por N. Wirth. Em estudo recente comparando algoritmos de geração de código e densidade do código para diferentes arquiteturas [WIRTH 86] mostrou que o tamanho do código gerado pelos compiladores para várias arquiteturas difere por um fator de 3. A densidade do código varia entre 1 e 4. Uma importante conclusão deste estudo, entretanto, é que muito da complexidade do compilador não é, necessariamente, devido a complexidade das instruções, mas sim decorrente de irregularidades inerentes das arquiteturas. Este fato tornou-se particularmente óbvio durante a comparação do código gerado para os processadores *Motorola 68000* (*MC68000*) [MOTOROLA 84] e do *National Semiconductor 32000 NS32000* [NS32000 84], os quais pertencem a categoria *CISC*. Uma irregularidade é estabelecida, por exemplo, permitindo alguns modos de endereçamento para algumas instruções enquanto para outras não, usando formatos diferentes para o conjunto de instruções, restringindo algumas operações para um subconjunto de registradores, etc.

Tendo em vista estes fatos, Wirth [WIRTH 87] sugere em seu trabalho a substituição da palavra *reduced* em *RISC* para *regular*. Regularidade segundo [WIRTH 87] no projeto de uma arquitetura deve aparecer como uma peça fundamental, porém ela sozinha não é suficiente. Ela deve ser acompanhada pela completeza. O conjunto de instruções deve refletir o conjunto básico de operadores disponíveis na linguagem. Baseados nestes dois aspectos das três arquiteturas analisadas, `NS32000`, `MC68000` e *Lilith* [OHRAN 84], `NS32000` representa um avanço sobre as mais recentes arquiteturas. Dada a irregularidade de `MC68000` o seu gerador de código é complexo e duas vezes maior que o código de `Lilith`.

Willian Wulf [WULF 81] vai mais longe e inclui a regularidade, a ortogonalidade e a capacidade de se compor como princípios fundamentais no projeto de arquiteturas, linguagens e compiladores. Um excelente exemplo de regularidade é a habilidade do compilador de tratar registradores e memória simetricamente tanto como fonte ou como destino, obtendo

um compilador simples e um bom código objeto. A ortogonalidade significa a habilidade de dividir a definição da máquina ou a definição da linguagem em um conjunto de assuntos separados e definir cada um deles isoladamente. Por exemplo, discutir tipos de dados, endereçamento e conjunto de instruções independentemente. Por capacidade de se compor é entendido a habilidade de compor, ortogonalmente, conceitos regulares de maneiras arbitrárias, se os princípios de regularidade e ortogonalidade foram seguidos. Por exemplo, pode ser possível utilizar todos os modos de endereçamento com todos operadores e todos os tipos de dados.

3 Conclusão

Este texto procurou apontar os problemas que existem nas arquiteturas RISC e algumas das soluções propostas para resolvê-los. Estes problemas ainda demandam melhores soluções.

Toda arquitetura *RISC* utiliza *pipeline* para executar as instruções mas o comprimento do *pipeline* e o método para remover *pipeline interlocks* variam. Como o ponto máximo de execução do *pipeline* é determinado pelo mais longo pedaço do *pipeline*, o importante é encontrar um ponto de equilíbrio entre as quatro partes da execução de instruções em uma arquitetura *RISC*: (1) carga de instrução (*fetch*); (2) leitura de registrador; (3) operações aritméticas e lógicas; (4) e escrita de registrador.

Com a filosofia destas novas arquiteturas esperava-se uma redução na complexidade dos compiladores. O obstáculo, entretanto, é uma densidade inferior de código. Questiona-se que um código expandido é bem aceito porque o tamanho de memória já não é mais um empecilho e, também, que o tempo adicional gasto para a carga de instrução pode ser compensado pelo uso de concorrência através de *pipelining*. Entretanto, conjectura-se que se o tamanho do código gerado para um programa em máquinas RISC for muito grande, este fato pode afetar negativamente o desempenho da memória [TANENBAUM 78].

As tentativas de soluções para os problemas das restrições impostas pela presença ou não de *interlocks* no *hardware* ainda deixam muito a desejar. As soluções são na maioria das vezes baseadas em heurísticas. [BERNSTEIN 89], por exemplo, apresenta uma solução quando o tempo máximo de atraso de uma instrução devido às restrições do *pipeline* é de 1 ciclo de máquina. A existência de um algoritmo de tempo polinomial para atrasos maiores que dois ciclos de máquina é uma questão em aberto e segundo [BERNSTEIN 89] o problema parece intratável. [HENNESSY 82, HENNESSY 83] também propõem uma solução heurística para este problema e mostra que o problema básico de reordenação de código de instrução no nível de máquina durante a compilação é NP-completo.

Outras arquiteturas classificadas como arquiteturas paralelas também merecem atenção de-

vido aos problemas existentes. Por exemplo, as arquiteturas horizontais, também denominadas *Array Processors* [WEISS 89], são aquelas arquiteturas que contêm várias unidades funcionais que podem operar concorrentemente, e onde cada unidade possui seu próprio barramento para conter os resultados das operações. Nas arquiteturas VLIW "*Very Long Instruction Words*", compiladores sofisticados provêem soluções tradicionalmente efetuadas por *hardware*. Os objetivos principais destas duas arquiteturas, do ponto de vista de *software*, são traduzirem o paralelismo de um programa, normalmente, codificado em uma linguagem de alto nível, no paralelismo em operações do *hardware*. A otimização de código é limitada contudo pela quantidade de código seqüencial em um dado programa. Estas arquiteturas baseiam-se essencialmente em escalonamento de código efetuado manualmente ou pelo compilador. A tarefa do escalonador é bem complexa e o código resultante nem sempre é eficiente.

Uma diferença significativa entre as arquiteturas horizontais e as demais é a ausência de *hardware interlocks* nas arquiteturas horizontais. Isto reduz o controle do *hardware* mas aumenta substancialmente a complexidade do escalonador de código.

Em resumo, os mais relevantes problemas interligados às arquiteturas RISC relacionados com a geração e otimização de código que merecem estudos são: (1) a interdependência entre a alocação de registradores e o escalonamento de instruções; (2) atrasos causados pelas instruções de desvios; (3) a presença de restrições impostas pelos processadores *pipelined*, como, por exemplo, a presença ou não de *interlocks* por *hardware*; (4) o problema de ordenação entre fases.

4 Referências

[AHO 86] AHO, A. V., ULLMAN, J.D. & SETHI R., "Compilers Principles, Techniques, and Tools", *Addison-Wesley, Reading*, MA, 1986.

[BERRY 78] BERRY, R. E., "Experience with the Pascal P-compiler", *Software-Practice & Experience*, 8,5 (September 1978).

[BENITEZ 88] BENITEZ, Manuel E. & DAVIDSON Jack W., "A Portable Global Optimizer and Linker", *SIGPLAN Notices* Volume 23, Number 7, July 1988.

[BERNSTEIN 89] BERNSTEIN, David & GERTNER, Izidor, "Scheduling Expression on a Pipelined Processor with a Maximal Delay of One Cycle", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 1, January 1989.

[COFFMAN 72] COFFMAN, E. G., Jr., & GRAHAM, R. L.,"Optimal Scheduling for Two-processor Systems", *Acta Inf. 1(1972)*, 200-213.

[COUTANT 86] COUTANT, Deborah S., HAMMOND, Carol L., & KELLEY Jon W., "Compilers for the New Generation of Hewlett-Packard Computers", *HEWLETT-PACKARD JOURNAL*, Vol. 37, No. 1, January 1986.

[DAVIDSON 81] DAVIDSON, Jack W.,"*Simplifying Code Generation Through Peephole Optimization*", Ph.D dissertation. Department of Computer Science, The University of Arizona, Tucson, Arizona, December 1981.

[DAVIDSON 84] DAVIDSON, Jack W., & FRASER, Christopher W., *Code Selection through Object Code Optimization*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984.

[DAVIDSON 84b] DAVIDSON, Jack W., & FRASER, Christopher W., "Register Allocation and Exhaustive Peephole Optimization", *Software-Practice and Experience*, Vol. 14(9), 8857-865. (September 1984).

[DAVIDSON 85] DAVIDSON, Jack W., "Simple Machine Description Grammars", *Computer Science Technical Report 85-22*, November 26, 1985, School of Engineering and Applied Science, Charlottesville, Virginia.

[DAVIDSON 86] DAVIDSON Jack W., "A Retargetable Instruction Reorganizer", *Proceedings of the ACM SIGPLAN'86, Symposium on Compiler Construction*, SIGPLAN NOTICES, Vol. 21, No. 7, July 1986.

[DAVIDSON 87a] DAVIDSON Jack W., & GRESH Joseph V., "Cint: A RISC Interpreter for the C Programming Language", *Proceedings of the Sigplan'87 Symposium on Interpreters and Interpretative Techniques*, St. Paul, Minnesota, June 24-26, 1987, SIGPLAN Notices Vol. 22, No. 7, July 1987.

[DAVIDSON 87b] DAVIDSON Jack W., & VAUGHAN Richard A., "The Effect of Instruction Set Complexity on Program Size and Memory Performance", *ASPLOS-II Proceedings - Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California October 5-8, 1987.

[DAVIDSON 90] DAVIDSON Jack W., & WHALLEY David B., "Reducing the Cost of Branches by Using Registers", *The 17th Annual International Symposium on Computer Architecture*, May 28-31, 1990 Seattle, Washigton, ACM SIGARCH COMPUTER ARCHITECTURE NEWS, Volume 18, Number 2, June 1990.

[DENNING 81] DENNING P. J.,"Computer Architectures:Some Old Ideias that haven't Quite Made It Yet",*CACM 24,9* (September 1981) 553-554 *ACM President's Letter.*

[DITZEL 87a] DITZEL, D. R., McLellan, H. R.,"The Hardware Architectural of the CRISP Microprocessor", *Proceedings of the 14th Annual Symposium on Computer Architecture*, Pittsburg, PA, June 1987,309-319.

[DITZEL 87b] DITZEL, D. R., McLellan, H. R.,"Branch Folding in the CRISP Microprocessor:Reducing Branch Delay to Zero",*Proceedings of the 14th Annual Symposium on Computer Architecture*, Pittsburg, PA, June 1987,2-9.

[GIBBONS 86] GIBBONS Phillip B., & MUCHNICK, Steven S., "Efficient Instruction Scheduling for a Pipelined Architecture", *Proceedings of the ACM SIGPLAN'86, Symposium on Compiler Construction*, SIGPLAN NOTICES, Vol. 21, No. 7, July 1986.

[GOODMAN 88] GOODMAN James R., & WEI-CHUNG-HSU, " Code Scheduling and Register Allocation in Large Basic Blocks", *International Conference on Supercomputing*, Conference ACM-PRESS Proceedings.St. Malo. France, July 4-8, 1988.

[HANSON 81] HANSON, D. R., "The Y Programming Languages", *SIGPLAN NOTICES 16.2(February 1981)*, 59-68.

[HAYES 88] HAYES, John P., COMPUTER ARCHITECTURE AND ORGANIZATION, "Parallel Processing", *McGROW-HILL BOOK COMPANY*, Second Edition, 1988.

[HENNESSY 82] HENNESSY, John, & GROSS, Thomas, "Code Generation and Reorganization in the Presence of Pipeline Constraints", *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 25-27, 1982.

[HENNESSY 83] HENNESSY, John, & GROSS, Thomas, "Postpass Code Optimization of Pipeline Constraints", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983.

[LEE 84] LEE, J. K. F. & SMITH, A. J.,"Branch Prediction Strategies and Branch Target Buffer Design",*IEEE Computer 17,1(January 1984)*,6-22.

[MOTOROLA 84] MOTOROLA Corp.,"MC68020 32-Bit Microprocessor User's Manual", *Prentice-Hall, Englewood Cliffs*, N. J., 1984.

[NS32000 84] National Semiconductor Corp., "Series 32000 Instruction Set Reference Manual", *National Semiconductor Corporation*, 1984.

[OHRAN 84] OHRAN, R. S., "Lilith and Modula-2", *BYTE 9,8(August 1984*, 181-192.

[PATTERSON 82] PATTERSON, David A., & SEQUIN, C. H.,"A VLSI RISC",*IEEE Computer 15,9 (September 1982)*, 8-21.

[PATTERSON 85] PATTERSON, David A., "Reduced Instruction Set Computers", *Communications of the ACM*, Vol. 28, No. 1, January 1985.

[RADIN 82] RADIN George, "The 801 Minicomputer",*Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47 (March 1982).

[TANENBAUM 78] TANENBAUM, A. S., "Implications of Structered Programming for Machine Architecture", *Communications of the ACM 21,3*, (March 1978) 237-246.

[WEISS 89] WEISS, Shlomo, "Scalar Supercomputer Architecture",*Proceedings of the IEEE*, Vol. 77 Number 12, December 1989 (1970-1982).

[WIRTH 86] WIRTH, N., "Microprocessor Architectures:A Comparison based on Code Generation by Compilers", *Comm. ACM, 29,10,(October 1986), 978-990.*

[WIRTH 87] WIRTH, N., "Hardware Architectures for Programming Languages and Programming Languages for Hardware Architectures", *ASPLOS-II Proceedings - Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California October 5-8, 1987.

[WULF 81] WULF W. A., "Compilers and Computer Architecture", *Computers 14,7* (July 1981) 41-48.