UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATÉMATICA, ESTATISTICA E CIÊNCIA DA COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

TÉCNICAS DE OTIMIZAÇÃO DE CÓDIGO PARA ARQUITETURAS RISC

por

Galileu Batista de Sousa[1]

29 de junho de 1992

[a]e-mail: galileu@dcc.unicamp.br

Técnicas de Otimização de Código para Arquiteturas RISC

Este exemplar corresponde a redação final da tese devidamente corrigida pelo Sr. Galileu Batista de Sousa e aprovada pela Comissão Julgadora.

Campinas, 29 de junho de 1992.

Prof. Dr. Tomasz Kowaltowski

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de MESTRE em Ciência da Computação.

Técnicas de Otimização de Código para Arquiteturas RISC

Galileu Batista de Sousa

Dissertação apresentada em 29 de Junho de 1992.

Banca Examinadora:

Dr. Tomasz Kowaltowski – UNICAMP – Orientador

Dr. Roberto da Silva Bigonha – UFMG

Dr. Ricardo de Oliveira Anido – UNICAMP

"Se um dia, já homem feito e realizado,
sentires que a terra cede aos teus pés,
que tuas obras se desmoronaram,
que não há ninguém à tua volta,
para te estender a mão,
esquece a tua maturidade,
passa pela tua mocidade,
volta à tua infância e balbucia
entre lágrimas e esperanças
as últimas palavras que sempre
te restarão na alma:
minha mãe, meu pai."

também dedico ao TD, Jr., Ciamel e, em especial, ao nosso pequeno Tiago; sem quaisquer restrições quanto à reorganização.

Agradecimentos

"O homem põe limites às trevas e explora até o extremo as grutas mais sombrias ... traz à luz o que está escondido. A sabedoria, porém, de onde é tirada? onde está a jazida da inteligência? só Deus a conhece ... mas ele disse ao homem: 'A sabedoria consiste em temer ao Senhor, e a inteligência está em afastar-se do mal'"

Jó - 28

Esta dissertação não é só minha. Ela é resultado do apoio técnico, financeiro e, sobretudo, emocional de várias pessoas ou instituições, pois 'muitos temores nascem do cansaço e da solidão'. Eu gostaria de lembrar especialmente algumas.

O Tomasz pelo apoio e paciência em alguns momentos críticos do tempo que trabalhamos juntos. Sua orientação e visão de computação foi, para mim, importante em inúmeros aspectos. A capacidade de contornar os meus problemas de comunicação também contribuiu para um resultado a contento.

Meus amigos de casa: Herbert *Hebão* Baier, Alfredo *Cocão* Jackson, *Zé* Guimarães e *Mestre* Léo foram parte importante para superar dificuldades e prover incentivos. Meu obrigado também por aturarem o meu mau humor, nem sempre condizente com o ótimo relacionamento que mantemos.

Com mais alguns amigos da graduação ainda compartilho vários sonhos e a esperança de um dia podermos realizá-los. Em especial estão José Leite Jr., Mário Sérgio Pinto e Ricardo Sobral; este cuidou com esmero dos meus interesses em Fortaleza enquanto estive por aqui.

O pessoal da FUNCEME sempre foi amigo e esteve pronto a ajudar; em particular, Francisco Lopes Viana e Manuel Pereira da Costa deram seu apoio em várias frentes, normalmente sem exigir coisa alguma em troca. A eles a minha profunda gratidão, que espero transformar em trabalho muito em breve.

Norma Suelly propôs o desafio de um mestrado. Meus agradecimentos não poderiam excluí-la.

Minha família sempre foi fonte constante de apoio. Minha Mãe é indescritível; é Mãe. Ela nunca compreendeu o porquê desta separação, porém sempre esteve pronta a ouvir meus argumentos. Meus irmãos sempre foram amigos e companheiros. Os primos: Johnson e Otacílio *Cabec* também são como irmãos. Tio Dema e Vô têm procurado ser, nos últimos oito anos, o pai que perdi, mas que está presente em todas as minhas reflexões. Aos demais familiares também agradeço a força.

Ciamel foi mais que eu esperava: sempre paciente e compreensiva, ela nunca criticou o meu desejo de recomeçar, mesmo quando já passara a hora de finalizar. Providenciar os momentos de diversão que me fizeram desligar dos problemas técnicos também ficou a seu cargo. Finalmente, sua dedicação a Tiago nestes últimos meses amenizou a saudade e fez com que eu pudesse confiá-la também o papel de Pai. Agora, espero retribuir em dobro o que ambos são e têm feito por mim.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - que deu o suporte financeiro à execução do Programa de Mestrado e à Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP - que viabilizou a parte final desta pesquisa.

Abstract

Reduced Instruction Set Computers (RISCs) offer higher performance through their very simple instruction repertoire and its efficient hardware implementation. On the other hand, studies of object code generated by compilers for high level languages have shown that the most frequently executed instructions are exactly the simplest ones. The improvement in performance of a general purpose computer is thus result of an integrated approach to compiler construction and architecture. These ideas have influenced design concepts in both areas.

In this dissertation we try to characterize these new architectures and their relation to compiling techniques, especially code optimization. Several aspects related to RISCs, such as register allocation and pipelines optimization are discussed. Some of the ideas discussed or proposed were tried within a production compiling system on SPARC (Sun Microsystems) architecture.

Sumário

Computadores com conjunto de instruções reduzido (RISC) buscam a maximização do desempenho oferecendo instruções muito primitivas e de implementação eficiente. Por outro lado, análises de código objeto gerado por tradutores de linguagens de alto nível mostram que as instruções mais freqüentemente executadas são também as mais elementares. O aumento do desempenho de um computador de uso geral é, portanto, resultante de uma abordagem integrada entre compiladores e arquitetura. Estes conceitos têm mudado parcialmente as métricas de projeto nas duas áreas.

Este trabalho consiste em caracterizar estas novas arquiteturas bem como seus relacionamentos com compiladores, ou mais precisamente, otimizadores de código. Aborda-se problemas fortemente identificados com o modelo RISC, tais como alocação de registradores e otimização de *pipelines*. Várias técnicas para tratar estes problemas são discutidas. Para validar as idéias descritas e/ou propostas, algumas das otimizações foram implementadas em um compilador de produção para a arquitetura SPARC.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

> *"However, if I had waited long enough I probably would never have written anything at all since there is a tendency when I really begin to learn something about a thing: not write about it ... every year I know there is more to learn, but I know some things which may be interesting now."*
>
> *Ernest Hemingway*

A avalanche de publicidade em torno de máquinas RISC – *Reduced Instruction Set Computers* – tem resultado em muita atividade científica e comercial em torno de projetos que integram fortemente arquitetura, compiladores e sistemas operacionais como meio de maximizar o desempenho de programas escritos em linguagens de alto nível.

Tecnicamente, a discussão concentra-se em determinar que métricas de arquitetura são mais adequadas para garantir maior desempenho de computadores de uso geral. A principal controvérsia está em torno da definição do conjunto de instruções oferecido pelo processador. RISCs se baseiam na capacidade dos compiladores gerarem código de boa qualidade para programas escritos em linguagem de alto nível, dado um pequeno conjunto de instruções que são executadas muito eficientemente. Em contrapartida, CISCs – *Complex Instruction Set Computers* – procuram deixar o conjunto de instruções mais próximo das linguagens de alto nível, facilitando, em tese, a conversão entre linguagens de alto e baixo nível.

RISC, entretanto, não é sinônimo apenas de conjunto de instruções reduzido. Instruções trabalhando unicamente sobre registradores (apenas LOAD/STORE acessam a memória), modos de endereçamento restritos e execução de instruções extremamente *pipelined* são outras características, igualmente importantes, encontradas no modelo.

A conseqüência da simplificação do *hardware* é uma série de questões a serem resolvidas pelo *software*. Operações sobre dados residentes em memória têm um alto custo (relativo) em máquinas LOAD/STORE, o que torna imperativa uma boa alocação de registradores, um problema reconhecidamente complexo no contexto de otimização de código. Similarmente, a execução de instruções em um *pipeline* pode gerar dependências de dados entre instruções, ou seja, uma instrução pode necessitar de dados que são calculados em instruções anteriores, mas que ainda não estão disponíveis. Existe também a possibilidade de operações triviais, do ponto de vista das linguagens, não estarem disponíveis no repertório, cabendo ao compilador (ou sistema operacional) sintetizá-las.

É exatamente a investigação do conjunto de problemas inerentes à compilação (ou precisamente otimização de código) para máquinas RISC o objetivo desta dissertação. Basicamente dois problemas são tratados em profundidade: alocação de registradores e otimização da execução *pipelined*. As próximas duas seções os introduzem em mais detalhes.

O primeiro problema que alguém imagina em um compilador que emite código para uma arquitetura com poucas instruções é a dificuldade de transformar a linguagem de alto nível em instruções muito primitivas. Curiosamente este não é o problema maior. A razão é que as linguagens intermediárias usadas pelos compiladores têm instruções também primitivas em pouco número. A emissão de uma instrução complexa resulta da combinação de várias das instruções intermediárias. Pode-se dizer, portanto, que o problema de escolher (selecionar) o código a emitir para uma máquina RISC é um caso particular da emissão de código para CISCs. Diante disto, o assunto não será tratado exaustivamente, embora uma das seções do capítulo 2 trate genericamente dos principais tópicos envolvidos com seleção de código.

1.1 Alocação de Registradores

Alocar registradores consiste em escolher determinadas variáveis do programa, ou temporários gerados pelo compilador, para permanecerem em registradores. O custo do acesso à memória em máquinas de conjunto de instruções reduzido torna este requisito determinante da qualidade do código gerado.

O princípio básico a ser respeitado por um algoritmo de alocação de registradores é que se duas variáveis podem ter seus valores (supõe-se distintos) ainda necessários a partir de um instante qualquer da execução do programa, então devem receber registradores distintos.

Um problema equivalente à alocação de registradores é a coloração de um grafo. Neste, quaisquer dois vértices devem receber cores distintas quando estão ligados por uma aresta. Se existir um modo de prever (e existe) que duas variáveis podem ter seus valores necessários em determinado instante da execução, então um grafo pode ser construído considerando as variáveis como vértices e determinando arestas a partir das "previsões". Deste modo uma cor pode também ser vista como um registrador, tornando os problemas equivalentes.[1]

Existem vários métodos para colorir um grafo associado a alocação de registradores e boa parte deste trabalho trata do assunto.

1.2 Otimização de *Pipelines*

Considere uma implementação RISC hipotética cujo *pipeline* tem três estágios: Busca (B), Execução (E) e acesso a operando em memória (M). Todas as operações aritméticas e lógicas terminam no segundo estágio, enquanto LOAD e STORE terminam no terceiro. Observe-se, então, os acontecimentos com a seqüência de instruções mostrada na Figura 1.1, quando executadas de forma *pipelined*.

No instante $t4$ ocorre um problema: o valor do registrador $r2$ é necessário, entretanto não se encontra disponível. Existe uma dependência de dados ou *interlock* entre instruções. Possíveis soluções para o problema são: introduzir uma instrução nop entre as instruções $i2$ e $i3$ com o

[1] A equivalência não resulta unicamente desta construção, mas (também) do comportamento que programas podem ter com relação ao fluxo de controle.

Rótulo	Instrução
i1	`add r1, r2, r3`
i2	`ld [r3], r2`
i3	`add r2, r3, r1`
i4	`and r4, r5, r6`

$t1$	$t2$	$t3$	$t4$	$t5$
(B)	(E)			
	(B)	(E)	(M)	
		(B)	(E)	
			(B)	(E)

Figura 1.1: Execução de uma seqüência de instruções no *pipeline*.

objetivo de retardar a terceira instrução; um outro esquema mais "inteligente" poderia perceber que a ordem de execução das instruções $i3$ e $i4$ é irrelevante no resultado final e, permutando essas instruções, a necessidade de nop deixa de existir. Essa última abordagem é chamada reorganização ou escalonamento de instruções, e as técnicas usadas para minimizar o número de nops também são abordadas em detalhes nesta dissertação.

1.3 Investigações

A interação entre arquitetura e compiladores freqüentemente dá margem a diferentes interpretações sobre a melhor forma de gerar código eficiente. A fim de comprovar alguns dos princípios que têm guiado o projeto de máquinas RISC, algumas investigações foram feitas usando o suporte do *Gnu C Compiler* – GCC [Sta89] para a SPARC [Sun87].

Em alguns casos, as experiências garantiram margem significativa de otimização sobre o código gerado pelo compilador, em outros confirmaram experimentos outrora executados em outros contextos e, por último, permitiram conjecturar sobre algumas das tendências de evolução das arquiteturas.

O GCC foi escolhido pela facilidade de acesso ao seu código fonte e por oferecer o suporte necessário à implementação de algumas otimizações. A versão 1.35 foi utilizada porque no momento que as investigações foram iniciadas, era a única ao alcance. Depois de conduzidos alguns experimentos, optou-se por continuar a utilizá-la, mesmo na existência de versões mais atualizadas. No contexto explorado, as versões mais recentes parecem ter pouco impacto sobre os resultados das experiências desenvolvidas, muito embora alguns *bugs* da versão utilizada tenham limitado os horizontes de experimentação. Uma nova versão (2.0), somente liberada na fase final de elaboração da dissertação, parece incorporar algumas dessas experiências.

1.4 Terminologia

A terminologia usada nesta dissertação geralmente coincide com a encontrada na literatura; termos em inglês são usados sempre que a tradução portuguesa não esteja padronizada. Nesta seção é introduzida a terminologia básica; quando necessário, esta terminologia é complementada no início de cada capítulo.

O termo *programa* denota um conjunto de procedimentos possivelmente compilados separadamente. *Procedimento* genericamente qualifica procedimentos, funções ou subrotinas de uma linguagem de programação particular.

Um *bloco básico* é uma seqüência de instruções sempre executada na totalidade e com, no máximo, uma instrução de desvio no final.

Adota-se genericamente o termo *variável* para caracterizar qualquer elemento alocável a um registrador, o que inclui variáveis escalares definidas pelo programador e temporários criados pelo compilador na geração de código para avaliação de expressões.

Uma variável ou é *global* ao programa ou pertence a algum procedimento, neste caso dita *local*. Os conceitos de local e global são, entretanto, bastante dependentes do contexto onde aplicados. Por exemplo, no capítulo 4, uma variável é considerada local se referenciada dentro de um único bloco básico e global, caso contrário; no capítulo 5 uma variável é tida como global se referenciada dentro de mais de um procedimento e local caso contrário. O contexto que precede o emprego de um ou outro termo é, em geral, suficiente para tornar claro o significado.

O *percentual de otimização* sobre um programa, normalmente expresso em termos de tempo, representa a fatia do tempo de execução original que foi eliminada em função do emprego de uma otimização. Define-se similarmente otimização sobre o tamanho do código e número de instruções, entre outras.

Em vários pontos do texto são apresentados exemplos de programas e suas respectivas *representações intermediárias*, ou mesmo o *código de montagem* final. Via de regra, a linguagem em que os trechos de programa são escritos é C [KR88]. Não serão precisamente definidas nem uma linguagem intermediária, nem uma linguagem de montagem, sendo o contexto suficiente para esclarecer o significado das (pequenas) porções de código; a única ressalva é que instruções sempre têm no último operando o destinatário da computação executada pela instrução.

Na literatura usa-se o termo *arquitetura* de uma máquina para expressar a visão que o programador (ou compilador) tem da máquina, ou seja, os seus aspectos externos: registradores, instruções, modos de endereçamento. Em alguns casos, esta visão é insuficiente para emissão de código de boa qualidade. No contexto estudado, arquitetura compreende os aspectos da *organização da máquina* que podem aumentar (ou diminuir) o seu desempenho, dependendo da forma de programação, por exemplo, acesso aos *caches* e organização de *pipelines*.

1.5 Os *Benchmarks*

Várias seções deste trabalho tratam com o comportamento estatístico de programas executados sobre uma arquitetura particular: a SPARC. Uma breve explicação sobre cada um dos programas e sobre suas respectivas origens é dada a seguir:

001.gcc-1.35: gcc-1.35 é o *Gnu C Compiler* versão 1.35 distribuído pela *Free Software Foundation*. A execução do programa consiste em converter 19 de seus próprios arquivos fontes em código de montagem otimizado para a Sun-3. GCC testa *cache*, entrada e saída e UCP. GCC é parte do *SPEC benchmark suite*.

008.espresso: espresso é uma ferramenta para geração e otimização de *Programmable Logic Arrays (PLAs)*. O tempo total de execução se refere a quatro modelos de entrada suportados pelo programa. O programa usa muita memória, testando assim o *cache* de dados. É parte do *SPEC benchmark suite*.

022.li: li é um interpretador Lisp. A execução determina o tempo gasto pelo programa para resolver o problema das nove rainhas. li testa a UCP e o alto número de chamadas de

procedimento é um desafio para máquinas com registradores organizados em forma de janelas. É parte do *SPEC benchmark suite*.

023.eqntott: eqntott transforma uma representação lógica de uma expressão *booleana* para uma tabela verdade. Exercita o *cache* de dados e a UCP. Também faz parte do *SPEC benchmark suite*.

stanford: stanford é composto por um conjunto de pequenos procedimentos que avaliam a UCP. Inclui torres de hanói, nove rainhas, transformada de Fourrier e *quick sort*. Foi obtido junto a John Hennessy da Universidade de Stanford;

navega: navega é um utilitário que faz mudança de projeção em imagens de satélites geoestacionários. Ele usa o posicionamento do satélite para localizar coordenadas geográficas em uma imagem matricial e, vice-versa. O programa faz uso de muitas operações de ponto flutuante. A sua execução consiste em encontrar a localização matricial de 10.000 coordenadas terrestres e, a seguir, fazer a operação contrária. Este programa é oriundo da Fundação Cearense de Meteorologia e Recursos Hídricos - FUNCEME;

reorg: reorg é um reorganizador de código de montagem para a arquitetura SPARC. Ele faz alguma entrada e saída, mas o seu tempo de execução é limitado pela UCP. Foi desenvolvido durante a elaboração deste trabalho. Seu tempo de execução corresponde a dez reorganizações do código de montagem referente a ele mesmo.

Todos os programas são escritos em linguagem C e as estatísticas foram obtidas utilizando o compilador GCC, versão 1.35.

Onde tempos de execução são considerados, eles se referem à melhor de quatro execuções em uma SPARC1+ com apenas uma *shell* executando. A unidade de tempo é o segundo e compreeende o tempo gasto no código do processo e no código do sistema operacional referente às chamadas de sistema feitas pelo processo. O utilitário *time* do sistema UNIX foi usado nas medidas.

1.6 Organização da Dissertação

A primeira parte desta dissertação discute arquiteturas RISC. O capítulo 2 oferece uma visão do modelo, partindo das motivações que culminaram com a ruptura do modelo CISC e explorando as ramificações que surgiram das propostas iniciais do padrão RISC. Este capítulo também oferece os conceitos básicos que são usados nas demais partes do trabalho. Capítulo 3 é um estudo de caso de uma arquitetura RISC, a SPARC. Além de apresentar a visão prática do modelo RISC, vários aspectos de interação entre *hardware* e *software* são apresentados, alguns deles explorados no capítulo 7.

A segunda parte trata de alocação de registradores. Várias técnicas, a maioria não aplicável exclusivamente a arquiteturas RISC, são estudadas. O capítulo 4 trata daquelas que alocam os registradores tendo a visão de apenas um procedimento por vez, as mais comumente usadas na prática. O capítulo seguinte mostra alguns contextos onde a alocação de registradores a um programa inteiro é importante e como ela pode ser feita. Algumas estratégias de alocação de registradores executadas pelo *hardware* são apresentadas também no mesmo capítulo.

A terceira parte discute reorganização de instruções. Inicialmente no contexto de otimização de pipelines, depois no sentido de encontrar paralelismo de granularidade fina, a fim de "alimentar"

máquinas RISC com mais de uma unidade funcional que operam paralelamente entre si. O capítulo 6 é dedicado a estes problemas e suas respectivas soluções.

A quarta e última parte relata experiências com o estudo e a implementação de algumas técnicas, discutidas nos capítulos precedentes, em um compilador de produção para a SPARC. De certa forma as informações oriundas das implementações estão distribuídas em vários capítulos, mas é no capítulo 7 que os resultados são apresentados em detalhes. O capítulo 8 conclui a dissertação com uma discussão dos trabalhos futuros.

Capítulo 2

Arquiteturas RISC

"...e aquilo que neste momento se revelará aos povos, surpreenderá a todos não por ser exótico, mas pelo fato de poder ter estado oculto, quando terá sido o óbvio."

Caetano Veloso - Um índio

2.1 Introdução

A qualidade de uma arquitetura pode ser medida pela sua adequação ao suporte das aplicações e pelo desempenho do *hardware* usado na sua implementação. Um computador de uso geral é normalmente programado em linguagem de alto nível. Esta, por sua vez, traduzida até o conjunto de instruções por um compilador e, o código objeto, executado usando a base de um sistema operacional. Portanto, o suporte às aplicações baseia-se primordialmente no trabalho conjunto entre arquitetura, compilador e sistema operacional. Por outro lado, a efetividade da implementação da arquitetura é resultado da velocidade do *hardware* em função do custo para obtê-la.

Diante destas métricas é importante escolher bem o conjunto de instruções a ser oferecido pela arquitetura, visto que ele permeia o trabalho de cada uma das fases entre a codificação da aplicação em uma linguagem de alto nível e a obtenção dos resultados desejados.

Arquiteturas RISC são o resultado de criteriosas análises sobre como o conjunto de instruções é usado no decorrer desse processo, considerando a freqüência de uso de instruções em programas compilados, bem como o custo de implementação de cada uma delas. A observação desses conceitos conduziu a arquiteturas "enxutas" não só em número de instruções, mas também em formatos e modos de endereçamento, viabilizando um maior desempenho de aplicações através do acoplamento entre compiladores e arquitetura.

O conteúdo deste capítulo é o seguinte: a seção 2.2 revê alguns aspectos históricos sobre projetos de máquinas nas duas últimas décadas; as suas influências para o surgimento de RISC estão na seção seguinte. As características do novo modelo são apresentadas na seção 2.4. O trabalho conjunto entre compiladores e arquiteturas é discutido na seção 2.5. O debate RISC *versus* CISC está na seção 2.6. A seção 2.7 é sobre processamento de ponto flutuante em RISCs e a 2.8 sobre a próxima geração de máquinas RISC.

2.2 Evolução das Arquiteturas

As limitações tecnológicas fizeram os primeiros computadores muito simples. Por volta de 1965, com o lançamento do IBM/360, a IBM inaugurou uma nova era no projeto de computadores ao oferecer várias implementações da mesma arquitetura. A partir de um *hardware* básico, microcódigo foi maciçamente empregado para oferecer um conjunto de instruções bastante sofisticado. Com um poder de expressividade mais alto que o *hardware* [Pat85], o microcódigo acabara com os limites à potência dos conjuntos de instruções.

Não coincidentemente, nesse mesmo período, as linguagens de programação de alto nível já se firmavam. Reconhecidamente era impossível impedir a evolução das linguagens, uma vez que elas se mostravam como um horizonte à emergente crise do *software*. Os projetistas de computadores, reconheceram então, um *gap* semântico entre linguagens de alto e baixo nível [Tan90], tornando compiladores muito complexos e passíveis a erros.

A verdade é que o microcódigo tomou uma importância muito grande nos projetos de computadores, desde o seu surgimento (até hoje). O VAX 11/780, por exemplo, tem nada menos que 303 instruções, microcodificadas, com tamanho variando de 2 a 57 bytes [Pat85]. A tendência generalizada em torno de microcódigo e a criação de ferramentas para desenvolvê-lo levou projetistas de computadores ([CIEDJK85]) a:

- tornar novas arquiteturas compatíveis com modelos anteriores. Este compromisso gera uma forte pressão para aumentar conjuntos de instruções, formatos e modos de endereçamento;
- tentar reduzir o *gap* semântico através do suporte, em *hardware*, a construções presentes nas linguagens de alto nível. O efeito de implementar instruções complexas e a capacidade que os compiladores tinham de gerá-las foram freqüentemente ignorados;
- transferir funções de *software* para microcódigo. Sendo este (na época) dez vezes mais rápido que as memórias convencionais, se uma única instrução pudesse descrever várias operações executadas em *hardware*, então o acesso à memória e o tempo de execução seriam drasticamente reduzidos.

Em meados dos anos setenta existia um conjunto de princípios a ser seguido por qualquer nova arquitetura. Esses incluíam o uso de microcódigo em larga escala e a minimização do tamanho dos programas, não sendo raras as máquinas projetadas para suportar uma linguagem ou modelo que execução específico. Tanenbaum [Tan90] cita um pesquisador da época: "As máquinas do futuro terão instruções com até seis campos e nenhum registrador, reduzindo largamente o *gap* semântico".

2.3 As Origens de RISC

Motivados pelo baixo aumento de desempenho proporcionado pelos avanços na tecnologia de *hardware*, pesquisadores começaram a fazer medidas para determinar como estavam sendo utilizadas as arquiteturas e linguagens de programação.

Do ponto de vista de linguagens uma grande variedade de programas foi analisada. Uma parte dos resultados ([Tan90]) é mostrada na Tabela 2.1. XPL e SAL são linguagens com estilo PL/I e Pascal, respectivamente. A média é aritmética e tem sentido apenas qualitativo, desde que diferentes conjuntos de programas foram analisados em cada caso. [Tan78] também examinou

Construção	SAL	XPL	FORTRAN	C	Pascal	Média
Atribuição	47	55	51	38	45	47
Condicional	17	17	10	43	29	23
Chamada	25	17	5	12	15	15
Repetição	6	5	9	3	5	6
Desvio	0	1	9	3	0	3
Outros	5	5	16	1	6	7

Tabela 2.1: Freqüência percentual de construções em linguagens de alto nível.

outras três características do código fonte de um sistema operacional escrito em SAL. Os resultados estão na tabela 2.2.

	0	1	2	3	4	≥ 5
(a) Termos	-	80	15	3	2	0
(b) Parâmetros	41	19	15	9	7	8
(c) Var. Locais	22	17	20	14	8	20

Tabela 2.2: Freqüência percentual de: (a) Termos por atribuição; (b) Parâmetros passados por chamada de procedimento; (c) Variáveis locais por procedimento.

Do ponto de vista do conjunto de instruções, as medidas são igualmente expressivas. No IBM 370, dez instruções (as mais simples) são responsáveis por 67% das execuções [Muc90a]. No VAX, os quatro modos de endereçamento básicos são empregados em 92% dos casos [Wie82]. Números similares também foram posteriormente encontrados para outras arquiteturas, incluindo o IBM-PC [AZ89].

As conclusões desses estudos foram:

- Os programas podem ser muito complexos, mas normalmente possuem uma estrutura muito simples;
- Os compiladores não conseguem fazer o uso apropriado das instruções possantes oferecidas pela arquitetura. Na prática, dadas uma linguagem e uma arquitetura existem mais diferenças que semelhanças entre elas.

Os resultados das medidas sobre o comportamento dos programas, a freqüência de uso de instruções, o surgimento das memórias *cache* e o conhecimento do princípio de localidade de referências dentro dos programas determinaram mudanças nos padrões já estabelecidos. Argumentou-se que a tentativa de eliminar o *gap* semântico só serviu para introduzir um *gap* de desempenho. Tornou-se evidente que para fazer uma arquitetura de uso geral o mais eficiente possível era necessário:

- Priorizar a implementação eficiente das instruções mais executadas;
- Minimizar o tempo de ciclo de relógio do sistema. O que demanda reduzir e balancear os tempos das seguintes atividades:

- Decodificação de instruções;
- Acesso aos registradores;
- Acesso ao *cache*;
- Operação da Unidade de Aritmética e Lógica.

- Minimizar o número de ciclos para executar cada instrução; ou menos rigorosamente, o número de ciclos para que os resultados de uma instrução estejam disponíveis à seguinte.

Projetistas de arquiteturas RISC tornaram mais forte a última regra ao exigirem que "todas" as instruções executem em apenas um ciclo.

A principal vantagem de arquiteturas RISC sobre arquiteturas CISC convencionais é a eliminação do efeito da implementação de instruções complexas sobre instruções simples e primitivas. Em uma máquina com grande conjunto de instruções é muito mais difícil traduzir as regras acima em resultados práticos, a custo baixo. A razão disso é o grande número de variáveis envolvidas, por exemplo, múltiplos tipos de dados, necessidade de acessos simultâneos à memória, várias combinações de formatos e muitos modos de endereçamento.

2.4 Princípios de Projeto em RISC

RISC baseia-se em extrair eficiência e funcionalidade da simplicidade. A evolução dos compiladores e a simplificação da arquitetura fizeram com que os resultados surgissem rapidamente. A rigor, o número de instruções é mais conseqüência do que causa. A razão principal da simplicidade do conjunto é a busca da eficiência. Entre as características do modelo estão:

- Instruções de um ciclo de relógio;
- Poucas instruções e modos de endereçamento;
- Arquitetura LOAD/STORE;
- Grande conjunto de registradores;
- *Pipelining* em alta escala;
- Eliminação do microcódigo;
- Formato fixo de instruções;
- Interdependência entre arquitetura e compiladores.

Note-se, entretanto, que estas características não são totalmente independentes.

2.4.1 Instruções de um ciclo de relógio

O que realmente distingue uma máquina RISC de uma máquina CISC não é o número de instruções, mas a natureza das instruções presentes no repertório. Embora sendo verdade que o número de instruções RISC seja pequeno, o mais importante é que quase a totalidade delas executa em um ciclo.

Para minimizar o tempo de ciclo de relógio, Patterson [Pat85] sugere que nunca se deve colocar uma nova instrução no repertório básico se não há uma forte razão para fazê-lo. Aqui, uma forte razão é função da possível freqüência de uso e do custo de implementação da instrução em questão. Portanto vale a regra: se uma instrução aumenta o tempo de ciclo em 10%, sua inclusão deve proporcionar, no mínimo, uma redução equivalente no número de instruções executadas.

2.4.2 Poucas instruções e modos de endereçamento

A ênfase em instruções de tempo de execução similar e implementação eficiente levou a maioria das implementações RISC a terem menos de 100 instruções e dois ou três modos de endereçamento.

Na prática, o pequeno e peculiar conjunto não é uma restrição muito forte, desde que as instruções presentes são as mais comumente executadas. Um exemplo é a ausência de instruções de multiplicação e divisão na maioria dos RISC. Embora primitivas, estas operações são muito caras quando comparadas às demais. Nestas situações é importante o poder de síntese dos compiladores. [Muc90b] relata que, em geral, essas operações têm, no mínimo, um operando constante, podendo ser implementadas eficientemente através de *shifts*, adições e subtrações [Muc90b]. Para a multiplicação, isto resulta um tempo médio de execução em "*software*" de apenas seis ciclos, comparados aos quase sessenta para o caso geral.[1]

Seguindo uma cadeia de vantagens, o reduzido número de instruções simplifica o projeto de uma unidade de execução *pipelined* [HB84] e diminui o custo do estágio de decodificação, além de liberar mais área na pastilha para implementação de um número maior de registradores ou *caches* internos.

2.4.3 Arquitetura LOAD/STORE

Minimizar o tempo de ciclo e executar em um único ciclo instruções que fazem acesso à memória são objetivos incompatíveis. A razão é relativamente óbvia: para cada operando em memória, um ciclo é gasto no cálculo do endereço e pelo menos um outro no acesso à memória (ou *cache*).[2] A solução normalmente adotada é permitir que apenas as instruções LOAD e STORE tenham operandos em memória, as demais operam unicamente sobre registradores ou operandos constantes imediatos. Esta restrição simplifica o problema, que pode agora ser resolvido com pequenas penalidades na implementação do *pipeline*, conforme detalhado nas próximas subseções.

[1] Novas versões das arquiteturas MIPS [Cho88] e SPARC [Sun87] oferecerão instruções de multiplicação e divisão em *hardware*. A idéia é ter uma espécie de execução em *background* da instrução e bloquear o acesso ao registrador alvo até que o resultado tenha sido, de fato, gerado. Um registrador especial, *scoreboarding*, contem um *bit* para cada registrador da arquitetura, indicando se o acesso a determinado registrador bloqueia ou não o processador.

[2] Salvo menção em contrário, memória RAM e *cache* serão usadas de forma intercambiável.

2.4.4 Grande conjunto de registradores

Se programas executando em um modelo LOAD/STORE (registrador-registrador) são submetidos a um alocador de registradores, o tráfego de (para) a memória pode ser grandemente reduzido, desde que o bom uso de registradores permite a reusabilidade de operandos. No entanto, o número de registradores na arquitetura deve ser suficiente para permitir uma boa alocação. O número típico de registradores encontrado é 32 (vide apêndice A).

Um dos primeiros RISC$, RISC I [PS82], do qual a SPARC é descendente, adotou uma solução elegante para evitar salvar e restaurar o contexto dos registradores durante chamadas de procedimento. O arquivo de registradores é dividido em conjuntos de tamanho fixo, com apenas um deles ativo em qualquer instante, representando os registradores conhecidos no momento. Quando uma chamada de procedimento ocorre, um novo conjunto (janela) é ativado e a janela anterior mantida intacta. No retorno, a janela original passa a ser a ativa. Na SPARC os deslizamentos são feitos usando instruções específicas, normalmente executadas no início e fim de cada procedimento. Detalhes no próximo capítulo.

Para permitir passagem de parâmetros em registradores, os conjuntos são organizados em uma fila circular e vizinhos têm alguns registradores em comum onde o chamador pode armazenar valores para uso do chamado. Alguns registradores globais são normalmente disponíveis. A Figura 2.1 ilustra um conjunto dividido em quatro janelas; w_i **loc** refere-se aos registradores disponíveis para variáveis locais na *i-ésima* janela. Similarmente os registradores em w_i **in** e w_i **out** são destinados à alocação de parâmetros formais e efetivos, respectivamente.

Eventualmente, após um conjunto de chamadas, nenhuma janela está livre. Se uma nova chamada acontece, então os registradores da janela menos recentemente utilizada são armazenados na memória, tornando-a disponível para uso pelo procedimento chamado. A restauração dos registradores pertencentes a uma janela é feita sob demanda [TS83].

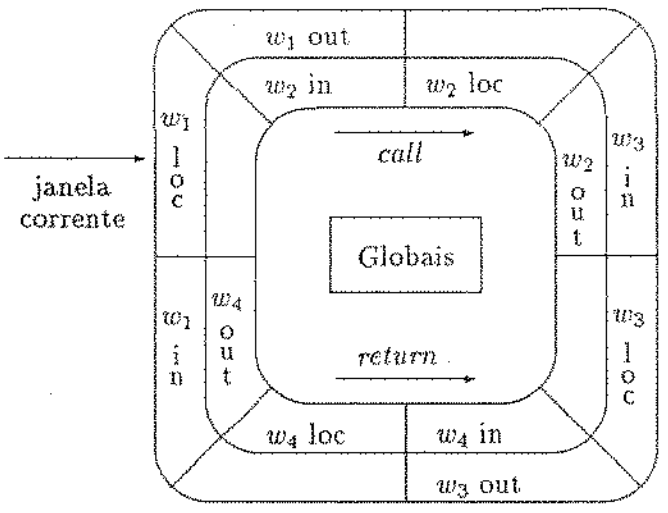


Figura 2.1: Janelas de Registradores.

2.4.5 *Pipelining* em alta escala

Execução em um ciclo não implica necessariamente que cada instrução permanece na UCP por apenas um ciclo. Rigorosamente, a execução total de qualquer instrução toma mais de um ciclo. Como já discutido, o acesso à memória gasta um ciclo e toda instrução antes de ser submetida para execução deve ser buscada na memória. Em geral, o processamento de uma instrução, numa máquina RISC, é composto de quatro estágios: busca, decodificação, execução e armazenamento de resultados. Execução em um ciclo refere-se ao tempo gasto no terceiro estágio do processamento da instrução.

Desde os primeiros projetos de computadores, *pipeline* tem sido uma das principais técnicas para melhorar o desempenho. A profundidade (número de estágios) do *pipeline* dá o aumento potencial da taxa de processamento. Mas um *pipeline* só é efetivo se ele pode ser mantido cheio e livre de inconsistências a maior parte do tempo. Neste sentido a profundidade atua apenas como um multiplicador ideal, não real, para o aumento do desempenho que pode conceitualmente ser obtido com a sobreposição temporal do processamento das instruções.

Vários fatores contribuem para limitar o desempenho de um *pipeline*, entre eles dependência de dados, instruções que alteram o fluxo seqüencial do programa e, também, o próprio custo de gerenciamento do *pipeline* em cada estágio [Hen84]. Dependência de dados é o termo para caracterizar a situação onde o processamento simultâneo de instruções no *pipeline* gera inconsistências em relação à execução de uma instrução por vez. Instruções que alteram o fluxo de controle criam a necessidade de recarregar estágios do *pipeline* que pressupõem execução estritamente seqüencial. O gerenciamento de *buffers* entre estágios e a prevenção e tratamento de dependências e desvios aumentam o tempo básico de cada estágio e por conseguinte diminuem o desempenho ideal do *pipeline*.

Nos exemplos a seguir, considere o *pipeline* simples, ainda assim realista, de três estágios:

Busca (B): uma nova instrução é buscada da memória;

Execução (E): a instrução buscada no ciclo anterior é decodificada e executada (no caso de LOAD/STORE o endereço é calculado);

Memória (M): se algum acesso à memória de dados é necessário, ele acontece neste estágio.

Apenas as instruções LOAD e STORE usam o terceiro estágio, as demais terminam um ciclo após serem buscadas.

Dependência de dados

Podem acontecer três tipos de dependências de dados no processo de execução *pipelined*. O primeiro e mais comum ocorre quando uma instrução requer um resultado calculado por uma outra instrução anterior, mas o resultado ainda não está disponível. Esta dependência é denominada *leitura-após-escrita* (RAW), ou dependência de fluxo.

O segundo, chamado *escrita-após-leitura* (WAR), ou anti-dependência, acontece quando um operando é atualizado antes que todas as instruções que necessitam de seu valor anterior o tenham lido.

Finalmente, uma depedência *escrita-após-escrita* (WAW) ou de saída pode resultar de uma alteração na ordem de atualização de operandos.

Observe que apenas o primeiro caso de dependência é verdadeiro, os dois últimos são, em princípio, conseqüência de reutilização de recursos. Obviamente, não existem dependências em função de múltiplas leituras. Para resolver dependências de RAW existem várias estratégias. Aborda-se, abaixo, duas por *software* e duas por *hardware*; uma combinação adequada delas sempre resolve o problema:

1. *Curto-Circuito* - Os resultados intermediários de qualquer computação em andamento são encaminhados aos demais estágios, se as instruções subseqüentes necessitam deles;

2. *Interlock* - Se o *hardware* deteta dependência de dados, alguns estágios do *pipeline* são congelados até que a dependência deixe de existir;

3. *No-op* - É o *interlock* por *software*; consiste em colocar instruções nulas (nop) entre instruções sempre que dependências RAW possam acontecer. Este esquema pode aumentar o tamanho estático do programa objeto e a complexidade do compilador, mas reduz a complexidade do *hardware*, aumentando a eficiência;

4. *Escalonamento de instruções* - É uma generalização da abordagem anterior, nesse caso o compilador se encarrega de reconhecer dependências estaticamente e mudar a ordem das instruções para eliminá-las.[3] A reorganização, entretanto, deve manter a ordem relativa entre algumas instruções de modo a deixar o resultado final das computações inalterado. Para executar essa tarefa o compilador deve conhecer o *pipeline* da máquina alvo. No capítulo 6 o assunto é discutido em detalhes.

Note-se que a ausência de *interlocks* em *hardware* obriga o uso de uma estratégia em *software*. A arquitetura SPARC, por exemplo, implementa parcialmente ambas as estratégias de *hardware* (1 e 2), ficando a cargo do compilador resolver algumas exceções e aumentar a eficiência do código (vide capítulo 3) através de escalonamento. A arquitetura MIPS não possui *interlock* e por isso, em conjunto com os compiladores, usa uma estratégia que abrange 1, 3 e 4.

Existe um conjunto de técnicas para projetar *pipelines* com número reduzido de dependências, mas o desempenho da arquitetura pode ser muito afetado [Nav90]. Em RISCs é possível simplificar o processo: para o caso de dependências WAR, uma solução simples é forçar que todos os dados necessários à execução de uma instrução sejam obtidos em uma fase de *pipelining* anterior a qualquer daquelas que atualizam dados. Para dependência WAW é suficiente exigir que todas as operações terminem em um único estágio. Curto-circuito é em geral muito efetivo no tratamento de dependências RAW quando o *pipeline* tem poucos estágios.

Instruções de acesso à memória

Se existem *caches* separados para dados e instruções e é possível fazer acessos simultâneos aos dois, então como resultado de iniciar uma instrução LOAD (STORE) no ciclo c, tem-se no ciclo $c+2$ a busca (armazenamento) do operando. Neste intervalo, uma nova instrução está sendo executada e outra buscada para execução. No ciclo $c+2$ duas instruções terminam. Portanto, muda-se parcialmente os objetivos, em vez de executar uma instrução a cada ciclo, inicia-se uma nova instrução a cada novo ciclo. Numa visão "macro" os resultados são equivalentes. Essa estrutura

[3] Pelo menos uma arquitetura, o CDC 6600, adotou esta estratégia em *hardware*.

de *pipeline* foi usada em uma máquina experimental da Hewllet-Packard [GM86]. Observe que o esquema cria algumas dependências de dados.

Por outro lado, se o *cache* não permite acesso concorrente a dados e instruções, então o *pipeline* é congelado por um ciclo para que o acesso à memória seja efetuado. Feito o acesso o fluxo continua normalmente. Grosseiramente este é o caso da arquitetura SPARC.

Estas duas abordagens dão origem a duas definições importantes que serão utilizadas nos capítulos 3, 6 e 7: custo de execução e latência de instruções. Define-se latência como o número de ciclos necessários para que os resultados de uma instrução estejam disponíveis à seguinte. O custo da instrução corresponde ao número de ciclos que a instrução ocupa com exclusividade o(s) estágio(s) de execução do pipeline, ou impede outras instruções de utilizá-lo(s).

Por exemplo, na primeira situação considerada para resolver o problema de LOADs (e STOREs), o fluxo de execução no *pipeline* continua inalterado, com uma instrução sendo executada em cada ciclo, assim o custo do LOAD é um ciclo, mas sua latência é também um ciclo, dado que seu resultado não pode ser utilizado imediatamente por uma instrução subseqüente. A segunda estratégia por outro lado, retarda o início da execução da instrução seguindo o LOAD em um ciclo, aumentando, por conseguinte, o seu custo também em um ciclo; não existe, porém, latência neste caso.

Como já notado, máquinas RISC têm, para a maioria da instruções, custo de execução de um ciclo e latência nula. Instruções LOAD e STORE aumentam uma das duas grandezas em função do acesso ao *cache*. Como será apresentado posteriormente, instruções de ponto flutuante são outra fonte de latências e custos acima de um ciclo.

Instruções de desvio

A outra fonte de problemas com *pipeline* diz respeito a instruções que alteram o contador de programa, doravante chamadas instruções de desvio.

Considere o trecho de programa na Figura 2.2 sendo executado no *pipeline* anteriormente apresentado. A fase de busca de instruções (B) baseia-se na ausência de desvios, por isto no momento que a instrução $i28$ está no estágio de execução (E) a instrução $i32$ já está sendo buscada. Após a fase de execução de $i28$, o contador de programa vale $i40$, mas $i32$ já foi buscada. Neste momento, o trivial seria anular a instrução $i32$; não é o que a maioria das arquiteturas RISC faz.

Rótulo	instrução
i24	add ...
i28	jmp i40
i32	or ...
i36	and ...
i40	xor ...

t_i	t_{i+1}	t_{i+2}	t_{i+3}
(B)	(E)		
	(B)	(E)	
		(B)	(E)
			(B)

Figura 2.2: Efeito da instrução de desvio na execução *pipelined*.

Sendo a freqüência de instruções de desvio alta, descartar a instrução seguindo o desvio pode reduzir consideravelmente o desempenho, a menos que alguma técnica de previsão de desvios seja considerada. Enquanto a previsão do alvo de um desvio pode, na grande maioria dos casos, ser feita com sucesso [LS84], ela não é suficiente quando atrasos no *pipeline* são indesejados. É necessário

algum mecanismo de iniciar a busca das instruções, que de acordo com a previsão sucedem o desvio, antes da execução (precisamente decodificação) deste. Ou seja, alguma espécie de *look-ahead* para verificar a presença de desvios deve ser executado; instruções de tamanho variável contribuem para complicar ainda mais a lógica de busca.

Um esquema alternativo e largamente difundido em máquinas LOAD/STORE é desvio com efeito retardado (ou *delayed-branch*). Neste caso um certo número de instruções (*slots*) seguindo o desvio será sempre executado. Cabe ao compilador encontrar instruções independentes da de desvio e deslocá-las para depois desta. Um simples otimizador *peephole* pode tratar eficientemente desvios incondicionais. O tratamento de desvios condicionais é um caso de escalonamento de instruções. Uma estratégia alternativa é implementada pela SPARC: a execução da instrução que segue o desvio pode ou não ser executada, baseada na resolução do desvio. Se a instrução não é executada, o ciclo é desperdiçado.

A alta freqüência de desvios é o maior obstáculo ao desempenho ideal do *pipeline*. Estatísticas [Rad82, Cho88] mostram que o número de desvios varia de acordo com o conjunto de instruções. No VAX 11/780, uma de cada quatro instruções executadas é um desvio tomado [Wie82]. Em arquiteturas LOAD/STORE o número de instruções de desvio (tomados ou não) é da ordem de 20%.

Usando um reorganizador para a arquitetura SPARC desenvolvido no decorrer deste trabalho (e apresentado no capítulo 7) coletou-se medidas estáticas para alguns programas com o fim de verificar o comportamento de desvios atrasados em uma arquitetuta particular. Os resultados estão na Tabela 2.3. Trabalhando com outros programas e um outro RISC, Hennessy [Hen84] também observou ser possível encontrar uma instrução adequada para preencher um *slot* em 80% dos casos, em média. Conclui-se que com a estratégia de *delayed-branch* adotada em RISCs, o efeito de desvios é comparável às melhores estratégias de prognosticar o fluxo de controle [LS84], a um custo de implementação muito baixo.

PROGRAMA	% DE DESVIOS	% DE SLOTS PREENCHIDOS
001.gcc-1.35	23,7 %	85,0 %
008.espresso	21,4 %	88,9 %
022.li	27,0 %	88,0 %
023.eqntott	23,2 %	85,7 %
navega	7,6 %	93,8 %
reorg	18,7 %	71,0 %
stanford	14,6 %	81,1 %

Tabela 2.3: Freqüência e reorganização (sobre código otimizado) de desvios na SPARC.

Virtualmente, tornar o *pipeline* muito profundo pode gerar um melhor desempenho, e portanto mascarar o efeito das instruções complexas sobre as simples. Na realidade o tempo de cada estágio do *pipe* é determinado pelo tempo gasto no maior estágio. Assim a lógica de decodificação para grandes conjuntos de instruções pode determinar o tempo dos demais estágios. Aumentar o número de estágios faz também aumentar o número de potenciais dependências de dados. Em CISCs este problema é até mais sério porque a maior expressividade das instruções torna a cadeia definição-uso [TS85] mais curta e densa.

Sumarizando: o relacionamento entre alteração do fluxo de controle e granularidade das ins-

truções mostra que *pipelines* profundos não necessariamente aumentam o desempenho de arquiteturas CISC. Isto é porque o número de operações (não instruções) por bloco básico é dependente da linguagem e da natureza da aplicação, mas independente do conjunto de instruções. O maior poder de expressão das instruções CISC leva o compilador (em alguns casos) a gerar menos instruções para mapear as operações. A situação é agravada porque, em geral, os otimizadores eliminam muitas das operações dentro dos blocos básicos, mas não reduzem proporcionalmente o número de desvios. A Tabela 2.4 mostra o número de instruções de desvios antes e após aplicar otimização (os números são relativos a medidas estáticas). Logo, um *pipeline* profundo pode ser pouco efetivo, mesmo porque quanto menor número de instruções entre desvios menores são os possíveis ganhos com reorganização de instruções.

	SEM OTIMIZAÇÃO			COM OTIMIZAÇÃO		
PROGRAMA	Desvios	Total	%	Desvios	Total	%
001.gcc-1.35	35.685	179.916	19,8	29.361	123.848	23,7
008.espresso	7.759	43.969	17,6	6.682	31.255	21,4
022.li	2.911	22.882	12,7	2.723	10.079	27,0
023.eqntott	1.296	6.421	20,2	1.051	4.532	23,2
navega	197	2.893	6,8	162	2.122	7,6
reorg	1.001	7.499	13,3	864	4.618	18,7
stanford	422	3.588	11,8	328	2.253	14,6

Tabela 2.4: Efeito de otimização sobre o número de desvios.

2.4.6 Eliminação do microcódigo

Além da tentantiva de diminuir o *gap* semântico, as métricas de arquitetura anteriores a RISC também objetivavam facilitar a programação de aplicações usando linguagem de máquina. Um conjunto de instruções de muito baixo nível poderia se tornar um grande obstáculo à programação de grandes sistemas, usando linguagem de montagem. Por outro lado, microcódigo é conveniente para implementar grandes conjuntos de instruções usando poucas primitivas, com as vantagens que uma abordagem hierárquica de múltiplos níveis propicia.

Do ponto de vista de desempenho, entretanto, estas idéias correspondem a deslocar tarefas de *software* para o *hardware*, sem necessariamente implicar em aumento da taxa de processamento. Para evitar o custo de interpretação de microcódigo, RISCs o eliminaram totalmente, ou seja, qualquer instrução é convertida diretamente em sinais de *hardware*. É possível seguir a mesma filosofia para máquinas com grandes conjuntos de instruções, mas os requerimentos em termos de área no *chip* e de tempo de projeto são muito maiores e os resultados, apenas, se equipariam àqueles dos RISCs.

2.4.7 Formato fixo de instruções

Tecnologias VLSI permitem que um grande número de portas lógicas seja colocadas em uma pastilha de silício, mas a velocidade de propagação de sinais é baixa, favorecendo atividades paralelas. No caso de RISC, a simplicidade da arquitetura conduz à simplicidade da implementação. A pouca

variedade de formatos permite decodificação paralela a baixo custo e reduz a probabilidade deste estágio ser o gargalo do *pipeline*.

Instruções de tamanho fixo (32 *bits*) e poucos formatos, enquanto favorecem projetistas de compiladores [Wul81], eliminam decodificações seriais encontradas em CISCs e fazem que o suporte a memória virtual seja bastante simplificado. Durante a busca de uma instrução, apenas um *cache miss* ou *page fault* pode ocorrer. Somente no caso de LOAD/STORE o mesmo pode acontecer durante a execução.

2.5 Interação Compiladores/Arquitetura

Há divergências sobre a dificuldade de construir compiladores para máquinas com conjunto de instruções reduzido. [Tan90] considera o compilador RISC mais complexo, enquanto [Cha82a] relata que teve seu trabalho simplificado quando construiu um compilador para uma arquitetura padrão RISC. A postura adotada aqui é que compilar para RISC tem seus próprios desafios e facilidades.

Algumas questões importantes surgem quando um compilador está gerando código para uma arquitetura RISC. Um compilador otimizador é normalmente dividido em duas partes: o *front-end* e o *back-end*. O primeiro converte o código fonte em uma linguagem intermediária de muito baixo nível que é processada (otimizada) pelo segundo; ao final, o código intermediário é mapeado para instruções de máquina (seleção de código).

O processo de seleção de código pode ser complicado. Se cada instrução da máquina alvo expressa mais de uma operação da linguagem intermediária (a determinado custo), o problema consiste em cobrir o grafo representando o programa, utilizando subgrafos que representam instruções. Em geral, existem várias seqüências de instruções que geram resultados corretos para o programa. A minimização do custo das instruções emitidas passa a ser função do tempo gasto pelo seletor de código. A interação entre o seletor e o otimizador é igualmente complexa porque uma otimização na linguagem intermediária pode provocar a emissão de instruções de maior custo, mesmo se o número de operações no formato intermediário é reduzido.

Em arquiteturas RISC, a interação entre otimizador e seletor é simplificada porque as instruções de máquina são mais próximas da linguagem intermediária, determinando poucas interseções entre operações executadas por instruções diferentes. Otimizações ambiciosas tendem a ser mais efetivas em máquinas RISC, pois toda transformação aplicada no código intermediário tem efeitos positivos no código objeto final.[4]

A natureza do conjunto de instruções também pode influenciar a qualidade do código gerado e a complexidade do seletor. Irregularidades na semântica das instruções, exigências de registradores específicos em certas instruções e endereçamento relativo a registradores fixos, só como exemplos, exigem análises adicionais para serem detetadas e implicam na emissão de código extra para atender as restrições (ou supostas facilidades). Sobre estes aspectos Wirth [Wir87] escreveu: "A complexidade do compilador não diz respeito ao número de instruções, mas à regularidade do conjunto," sugerindo inclusive que RISC signifique *REGULAR Instruction Set Computers*. Semelhantes opiniões podem ser encontradas em [Wul81, Den78].

Em segundo lugar está a exposição de detalhes de implementação, como estrutura de *pipelines* ao compilador. Isto caracteriza a sintonia entre *hardware* e *software* e tende a deixar este mais

[4] Isto não impede que algumas otimizações interfiram entre si.

complicado, contudo garante a maximização da eficiência de aplicações sobre a arquitetura. Para controlar a complexidade do otimizador, bem como aumentar a qualidade do código emitido, é necessário que o conjunto de instruções seja pequeno, mas que toda a informação envolvida na execução de instruções seja conhecida em tempo de compilação [Hen84].

A natureza LOAD/STORE destas arquiteturas exige boas técnicas de alocação de registradores a fim de diminuir o tráfego memória/processador na busca de operandos. Indiretamente estas técnicas têm o efeito de diminuir, também, o número de instruções buscadas e executadas.

Finalmente é importante ressaltar que o pequeno número de instruções pode exigir trabalho do compilador para sintetizar operações, ainda que primitivas. Esta abordagem pode também trazer vantagens considerando-se que, em alguns casos, os dados sob operação são parcialmente conhecidos em tempo de compilação (ver exemplos em [Rad82]). Observe, portanto, que é melhor ter instruções de mais baixo nível que de mais alto. Os compiladores da HP [CHK86], por exemplo, emitem código como se todas as operações do formato intermediário existissem como instruções. Um passo seguinte as converte para o código apropriado, otimizando o que for possível. Esta idéia, denominada *millicode*, mostrou-se muito eficiente na geração de código para a aritmética do COBOL, usando umas poucas instruções básicas da arquitetura.

Vários experimentos nas universidades de Berkeley e Stanford [Hen84] ratificaram as idéias descritas acima. Compiladores geram código de melhor qualidade para arquiteturas RISC. A qualidade do código é medida em função do que ele pode ser melhorado se cuidadosamente escrito a mão. Como nas experiências o mesmo compilador foi utilizado, mudando apenas a descrição da máquina alvo, conclui-se que é mais difícil conseguir bom código em máquinas de grandes conjuntos de instruções, principalmente quando estes são irregulares.

2.6 RISC *versus* CISC

Uma arquitetura RISC que executa instruções muito eficientemente só é atrativa se o número destas instruções não for excessivo em relação ao número de instruções CISC para executar a mesma tarefa.

Números apresentados em [Rad82] indicam que o número de instruções RISC é até 50% maior que o equivalente CISC para executar a mesma tarefa. Medidas estáticas mais recentes apontam um limite de apenas 20% [Muc90a]. Em qualquer caso, o reduzido número de ciclos por instrução é suficiente para sobrepor o mais alto número de instruções, em relação a CISCs [BC91].

A desvantagem potencial de um grande número de instruções executadas à taxa de uma a cada ciclo de relógio é a alta banda de passagem[5] de memória necessária. Todavia, desde que *caches* de instruções são apenas lidos, seu projeto é mais simples. O trabalho de Davidson [DV87] mostra que para acesso a instruções, um *cache* de 64 Kb tem a mesma taxa de acerto em RISCs e CISCs. Do ponto de vista de operandos, as arquiteturas LOAD/STORE têm menos requerimentos de memória quando os programas passaram por um alocador de registradores. Isto é importante porque o acesso a dados é menos previsível do que o acesso a instruções, diminuindo a eficiência do *cache*.

A grandeza mais difícil de remover quando comparando uma máquina RISC e uma CISC é a tecnologia usada na implementação. O que é mais razoável nestes casos é tornar comparáveis máquinas que possuem preços semelhantes.

Sem desejar apresentar conclusões definitivas tenta-se, a seguir, quantificar o desempenho de duas arquiteturas: O MC68020 (CISC) e a SPARC (RISC). O MC68020 gasta, em média, 7 ciclos

[5] do inglês: *bandwidth*

de relógio para executar uma instrução. Em um RISC típico, esse valor está em torno de 1,5. O primitivismo das instruções faz com que o número de instruções RISC para executar uma tarefa seja até 20% maior que o equivalente em CISC [Muc90a]. Independente do *benchmark* vale a relação:

$$T_e = \frac{N_i \times T_c}{C}, \quad onde:$$

- T_e - Tempo de execução de um programa;
- N_i - Número de instruções executadas;
- T_c - Tempo médio de ciclo de instrução (em ciclos de relógio);
- C - A freqüência do relógio da máquina.

Na ausência de informações específicas sobre um programa, essa relação e os dados acima, mostram que um processador RISC pode ser quatro vezes mais rápido que um CISC de mesma freqüência de relógio. Note-se que os resultados variam em função do *mix* de instruções no *benchmark* e ignoram atrasos devido ao *cache*. Considere-se também que o tempo de execução de um programa não depende exclusivamente da velocidade do processador.

2.7 Ponto Flutuante em RISC

Desde muito cedo ficou claro que introduzir operações de ponto flutuante em arquiteturas RISC era uma tarefa complexa. Por natureza, operações sobre números reais podem ter custo algumas vezes maior que sobre inteiros. Aumentar o número de estágios do *pipeline* ou o tempo de ciclo básico foi ignorado porque:

- Grande parte das aplicações não têm operações de ponto flutuante;
- O desempenho de aplicações de ponto flutuante é fortemente dependente de operações sobre inteiros (acesso à memória, cálculos de endereços). O programa **navega**, por exemplo, tem pouco mais 25% de suas instruções operando sobre números reais (medidas estaticamente).

O mecanismo mais simples de incluir instruções de ponto flutuante é através de um coprocessador (COP). Este dispositivo é passivo, no sentido que ele não inicia instruções. A unidade inteira (UI) ao encontrar uma instrução de ponto flutuante decodifica-a e repassa ao coprocessador. Uma vez iniciada a instrução no COP, a UI continua a execução de outras instruções.

No contexto RISC, os operandos do coprocessador estão sempre em registradores que se comunicam com a memória via instruções da UI. Dependendo da arquitetura, o conjunto de registradores pode ser globalmente compartilhado [Mel89] ou privativo a cada unidade [Muc90a].

Sincronização é necessária em alguns casos:

- O COP não pode aceitar a próxima instrução e, assim, bloqueia a UI. Isto acontece, por exemplo, quando o tempo de ciclo básico da UI e COP são diferentes;
- Uma instrução para armazenamento de um registrador chega à UI, mas o COP ainda não o gerou (RAW).

- Uma instrução para carregamento de um registrador chega à UI, mas o COP necessita do valor antigo daquele registrador (WAR).

A primeira situação é controlada em *hardware*. As duas últimas surgem da execução paralela e porque, independentemente do modelo, a UI também opera sobre registradores do COP, gerando dependências inter-unidades. Em um arquivo de registradores globalmente compartilhado as técnicas de curto-circuito e *interlock* (através de *scoreboarding*) podem ser facilmente estendidas. Em arquivos privativos as dependências são tratadas por sinais de controle entre as unidades.

Quando alto desempenho de ponto flutuante é requerido, coprocessadores têm desvantagens. A principal delas é que MIPS[6] e MFLOPS[7] tornam-se "grandezas inversamente proporcionais", pois todas as instruções de ponto flutuante são, em um certo sentido, instruções da unidade inteira. O *mix* de instruções não permite que nem a UI, nem o COP, atinjam sua taxa de processamento de pico. Estas limitações motivaram a segunda geração de máquinas RISC, descritas na próxima seção.

2.8 Segunda Geração de RISC

Quantitativamente o tempo gasto para executar uma instrução pode ser diminuído em função da tecnologia de *hardware* usada na implementação da arquitetura. Dada uma arquitetura, um aumento de desempenho qualitativo só é atingido quando o número de instruções por ciclo é aumentado através da exploração de paralelismo. A questão é saber se os programas apresentam paralelismo e se o custo de explorá-lo não excede as vantagens.

Várias tentativas, algumas bem sucedidas, de explorar paralelismo têm sido propostas. Entre elas pode-se citar: processadores vetoriais, multiprocessadores e máquinas dirigidas pelo fluxo dos dados. Processadores vetoriais são muito efetivos, mas restritos a determinadas aplicações. Multiprocessadores ainda apresentam desafios sobre sincronização e divisão de tarefas e máquinas de fluxo de dados pagam alto preço em programas estritamente seqüenciais.

Mais recentemente, dois novos modelos têm sido desenvolvidos para explorar paralelismo de baixa granularidade: VLIW[8] e Superescalares. Estas máquinas têm uma estrutura básica muito semelhante a arquiteturas RISC convencionais, mas são capazes de executar mais de uma operação por ciclo de relógio.

2.8.1 VLIW

Máquinas VLIW são projetadas para explorar o paralelismo de operações não relacionadas atuando sobre dados independentes. Existem várias operações codificadas dentro de uma instrução e um número fixo de unidades funcionais para executar cada uma destas operações. A principal característica é que cada unidade "olha" em uma porção fixa da instrução, executando a operação ali codificada. A Figura 2.3 serve para visualizar as demais características do modelo [Gas89]:

- Um conjunto de unidades funcionais;
- Um ou mais arquivos de registradores;

[6] Milhões de Instruções Por Segundo
[7] Milhões de Operações de Ponto Flutuante por Segundo
[8] *Very Long Instruction Word*

- Unidade de controle (e busca) central;
- Unidades funcionais *pipelined* e sincronizadas;

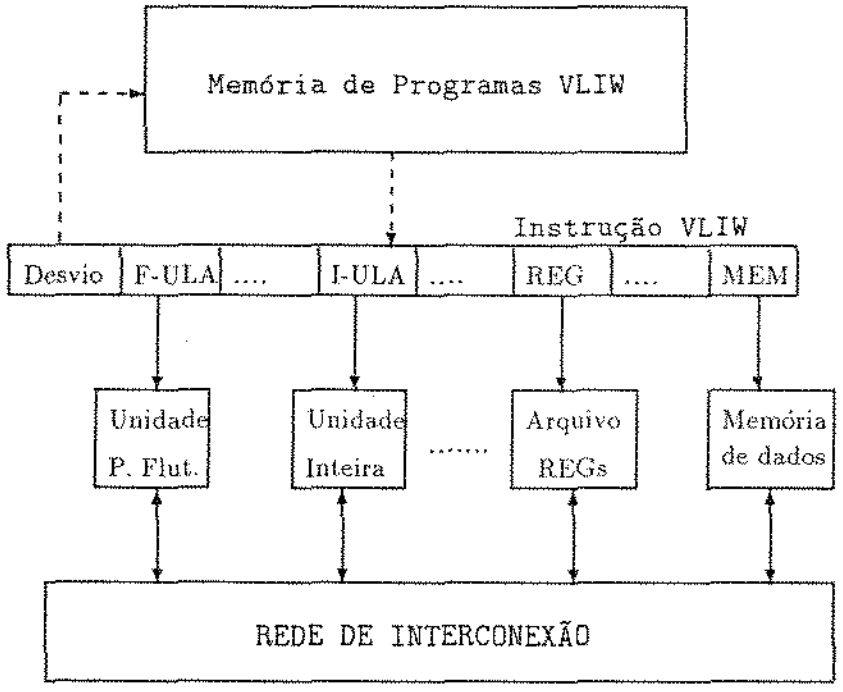


Figura 2.3: Diagrama de blocos de uma arquitetura VLIW genérica.

Estas características conduzem à seguinte semântica de operação:

- Em cada ciclo a unidade de controle busca uma nova instrução longa (tipicamente entre 100 e 1000 *bits*);
- Cada unidade funcional atua sobre uma parte fixa da instrução;
- Operações de baixa latência;
- Controle de acesso a recursos feito em tempo de compilação.

A semelhança com a filosofia RISC vem de uma série de fatores: modelo LOAD/STORE, *interlocks* em *software* e integração entre compiladores e arquitetura. Algumas instruções têm latência maior de um ciclo, conseqüência da natureza diversa das unidades funcionais e da necessidade de fazer as instruções primitivas mais eficientes.

Vários problemas surgem com uma implementação real de VLIW. O principal deles é que uma instrução muito longa aumenta a necessidade da banda de passagem do *cache* e pode diminuir

consideravelmente a densidade do código em programas estritamente seqüenciais (vide [CNO+88] para números sobre o código do UNIX).

Alguns mecanismos têm sido adotados para minimizar o tamanho do código em arquiteturas de instruções longas. No VLIW Multiflow [CNO+88], o código é armazenado na memória em blocos de 32 *words* com uma máscara de uma *word* indicando as palavras diferentes de nop, as únicas, de fato, armazenadas. Durante buscas da memória para o *cache* de instruções há uma expansão da informação, de maneira que o *cache* contém instruções no formato que o processador aceita. O Intel i860 [KM89], que têm uma unidade inteira e uma de ponto flutuante, dedica um dos *bits* de cada instrução inteira para indicar se a "próxima execução" consiste em uma instrução inteira e uma de ponto flutuante, caso onde duas instrução são buscadas. Ligando e desligando os *bits* das instruções inteiras é possível "ativar" a busca de instruções de ponto flutuante, ao mesmo tempo que nops não necessitam ser explicitamente representados.

Idealmente uma implementação deveria ter um único arquivo de registradores interconectado a todas as unidades funcionais. Esta característica simplificaria o trabalho do compilador e aumentaria a qualidade do código gerado, pois a seleção da unidade a executar uma operação estaria vinculada unicamente ao tipo de operação. A tecnologia de hoje não permite um número muito alto de portas de acesso a registradores, por isso existem conjuntos compartilhados apenas parcialmente, dimuindo a regularidade do conjunto de instruções.

Por outro lado, uma arquitetura VLIW só é efetiva quando suas unidades funcionais passam a maior parte do tempo ocupadas. A disponibilidade de um compilador com alto poder de reorganização de instruções passa a ser crítica, de outra forma pode haver uma explosão de código sem aumento de desempenho. Aqui, os requerimentos são mais fortes. Dependências dentro de uma unidade funcional *pipelined* devem ser evitadas, ao mesmo tempo que existe a necessidade de um escalonamento horizontal para alimentar as várias unidades. A alta freqüência de desvios é mais sensível em máquinas desta natureza. Algumas técnicas desenvolvidas para aumentar o tamanho dos blocos básicos e permitir maior eficiência, tais como: desmembramento de malhas[9], *trace scheduling* e *software pipelining* são apresentadas no capítulo 6.

2.8.2 Superescalares

Uma máquina superescalar de grau N pode executar até N instruções por ciclo de relógio, em paralelo. A operação exige, então, uma unidade de busca capaz de alimentar todas as unidades. Se um paralelismo de tal ordem não é disponível, pelo menos uma das unidades é bloqueada a espera de resultados das instruções anteriores ou de uma nova instrução.

Relacionamentos em relação a VLIW são:

- A decodificação de instruções em superescalares é mais complexa: instruções devem ser associadas às unidades apropriadas, o envio de instruções a cada unidade não pode exceder a capacidade desta. VLIWs, ao contrário, previnem estaticamente quaisquer excessos;
- A densidade do código superescalar não é afetada pela ausência de paralelismo;
- Ao contrário de VLIW, superescalares podem ser vistos como uma metodologia de implementação da arquitetura, permitindo compatibilidades em nível de código objeto com imple-

[9] do inglês: *loop unrolling*

mentações escalares. Em função desta compatibilidade, são o caminho natural para fabricantes de máquinas RISCs.

Exploração de paralelismo exige soluções de compromisso. Mesmo com as principais características de RISCs, estas máquinas têm que tratar com relacionamentos entre unidades funcionais, desempenho de computações de ponto flutuante, recursos compartilhados, além de exigirem sistemas de memória de mais alto desempenho. Com estas restrições, torna-se difícil obter um ciclo de relógio balanceado nas diversas atividades do processamento de instruções. Isto tem levado à proliferação de algumas facilidades ou instruções em relação ao repertório básico dos RISCs tradicionais, mas a filosofia geral continua muito presente nas arquiteturas mais recentes.

2.9 Conclusões

Desde que os primeiros RISC surgiram, a tecnologia de implementação de arquiteturas avançou em muito. Os 44.500 transistores presentes no processador RISC I são duas ordens de grandeza a menos do que é possível colocar hoje em uma única pastilha. Estima-se que a capacidade de integração seja duplicada a cada ano.[10] Como o surgimento dos RISCs foi parcialmente motivada pela integração dos circuitos, vendedores de máquinas CISCs [Cam91] têm argumentado que no futuro não existirão grandes diferenças de desempenho entre os modelos.

Poder-se-ia dizer que em última análise seria possível executar cada instrução CISC com uma circuitaria particular, eliminando microcódigo e levando as instruções simples a terem custo baixo. Mesmo que o custo de colocar um grande número de transistores em uma pastilha não seja significativo em tecnologias VLSI [Sny84], a forma de organizá-los pode ter impacto considerável na velocidade de processamento. Portanto, a sincronização entre as execuções de instruções complexas e simples, a lógica serial de decodificação e a replicação de operações em muitas instruções permaneceriam como "gargalos" em projetos CISCs. O projeto do pipeline continuaria igualmente complexo. No nível de implementação, tudo isto torna mais difícil diminuir o tempo de ciclo de relógio. Na prática grande parte da área continuaria dedicada a tratar situações pouco usuais, tudo que seria removido era o microcódigo. Em contrapartida, RISCs podem usar o espaço disponível nas pastilhas para aumentar a funcionalidade em processamento de ponto flutuante, aceleradores gráficos e caches maiores. O que, aliás, já pode ser encontrado em projetos recentes.

Para observar o estado atual da competição entre RISCs e CISCs, considere os dados sobre duas máquinas do mesmo fabricante (Intel), com mesma freqüência de relógio (33 MHz), implementadas com a mesma tecnologia e aproximadamente o mesmo número de transistores (1 milhão): O 80486 e o i860 [KM89]. Executando os SPEC *benchmarks*, o i860 é duas vezes mais rápido que o 80486. Considerando que o SPEC dá grande ênfase ao comportamento do cache e que o 80486 tinha (no teste) dez vezes mais *cache* que o i860, então os dados são significativos [SPE91].

Em resumo, como máquinas de uso geral, RISCs são qualitativamente superiores e possivelmente continuarão a oferecer melhor desempenho. Por esta razão fabricantes tradicionais de CISCs, como IBM, Intel e Motorola estão tomando o caminho dos RISCs.

[10] Respeitados os limites físicos.

Capítulo 3

Arquitetura SPARC

"Simplicity, I guess, is a way of saying it. I am all for simplicity. If it's very complicated, I can't understand it."
Seymour Cray

3.1 Introdução

A SPARC (*Scalable Processor Architecture*) é uma das mais populares arquiteturas RISC hoje (1992) no mercado. O termo *scalable* refere-se à capacidade da arquitetura acompanhar a evolução da tecnologia de fabricação, em virtude da sua simplicidade. O projeto da SPARC foi desenvolvido pela SUN Microsystems e descende diretamente dos primeiros RISCs projetados na Universidade da Califórnia em Berkeley.

As principais características da arquitetura são:

- Instruções executáveis, na sua grande maioria, em um único ciclo de relógio;
- Arquitetura LOAD/STORE;
- Poucas instruções e formatos. Todas as instruções têm 32 *bits* e são alinhadas em múltiplos de quatro *bytes* na memória;
- Grande conjunto de registradores, organizado em janelas deslizantes sobrepostas;
- Esquema de *delayed-jump* com execução condicional da instrução subseqüente;
- Unidade de ponto flutuante que pode executar concorrentemente com instruções operando sobre inteiros;
- Suporte a um coprocessador externo;

Este capítulo descreve a arquitetura SPARC [Sun87, BF90]. Detalhes de implementação são mencionados até onde podem interessar para o processo de geração de código. A SPARC foi escolhida, para estudo de caso, por sintetizar muitas das características dos RISCs de primeira geração, além de implementar o conjunto de registradores sob a forma de janelas. Alguns artigos descrevem outras arquiteturas: HP-PA [Lee89], MIPS-X [GHPR88], IBM-RS/6000 [Gro90, OG90,

HO91], Intel i860 [KM89] e Motorola 88000 [Mel89]. A próxima seção dá uma visão geral da organização e do conjunto de instruções da SPARC. A seção 3.3 trata da interação entre arquitetura, compiladores e sistema operacional para criar o ambiente de execução de programas. Alguns detalhes de implementação das máquinas SPARC atuais estão na seção 3.4. As oportunidades para reorganização de código estão na seção 3.5.

3.2 Visão Geral

Conforme esquematizado na Figura 3.1, a arquitetura é dividida em cinco componentes: a unidade inteira (UI) e a unidade de ponto flutuante (UPF) são as principais, cada uma com seu próprio conjunto de registradores. O *cache* é compartilhado para dados e instruções; os endereços armazenados são virtuais, tornando possível o cálculo do endereço físico, na unidade de gerenciamento de memória (UGM), paralelamente à busca no *cache*. Um coprocessador (COP) pode ser adicionado ao sistema, sua interação com os demais componentes é semelhante à da UPF. Os barramentos de dados e endereços são todos de 32 *bits*.

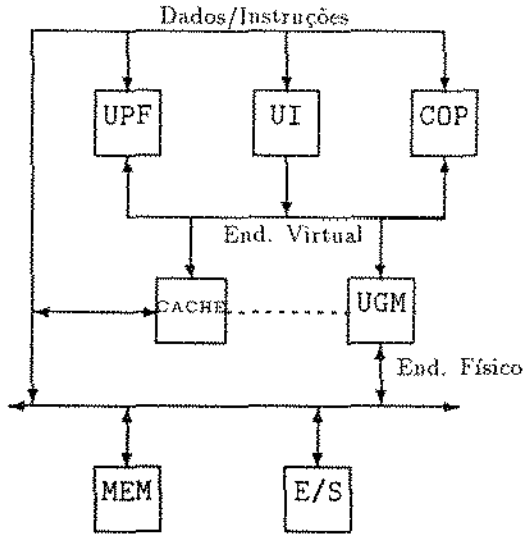


Figura 3.1: Diagrama de blocos da SPARC.

A UI é o elemento básico de processamento. Ela busca todas as instruções, executa as que operam sobre inteiros e controla o acesso à memória de dados, mesmo quando registradores da UPF são fonte ou destino de dados.

Instruções de ponto flutuante são executadas pela UPF. Quando a UI encontra uma instrução que opera sobre registradores de ponto flutuante passa-a à UPF e continua o seu fluxo normal de execução, buscando uma nova instrução.

Registradores

A UPF possui 32 registradores ($f0$-$f31$) de precisão simples (32 *bits*) usados exclusivamente em operações de ponto flutuante. Registradores são agrupados de dois em dois, e de quatro em quatro, para execução de operações de dupla precisão e precisão estendida, respectivamente.

Os registradores da UI são organizados em janelas deslizantes sobrepostas. Janelas têm 24 registradores de 32 *bits* ($r8$-$r31$), com sobreposição de oito ($r24$-$r31$) com a janela anterior e oito ($r8$-$r15$) com a posterior. Outros oito registradores ($r0$-$r7$) são globais a todas as janelas (vide esquema na Figura 2.1). O registrador $r0$ é iniciado em *hardware* para o valor zero.

Ambas as unidades têm registradores especiais contendo códigos de condição. Não há comunicação direta entre os registradores das duas unidades, movimentação é feita através da memória.

Instruções

As instruções aritméticas e lógicas se enquadram nos formatos (1a) e (1b) da Figura 3.2. O formato (1a) especifica dois registradores fonte e um destino; operações envolvendo um registrador e uma constante entre -4096 e +4096 são especificadas pelo formato (1b). A SPARC suporta adição, subtração, operações lógicas (e, ou, ou exclusivo), rotação e *shift* aritmético. Opcionalmente todas as instruções modificam o registrador de códigos de condição. Existe uma instrução que faz um passo de uma multiplicação completa (detalhes em [Sun87]).

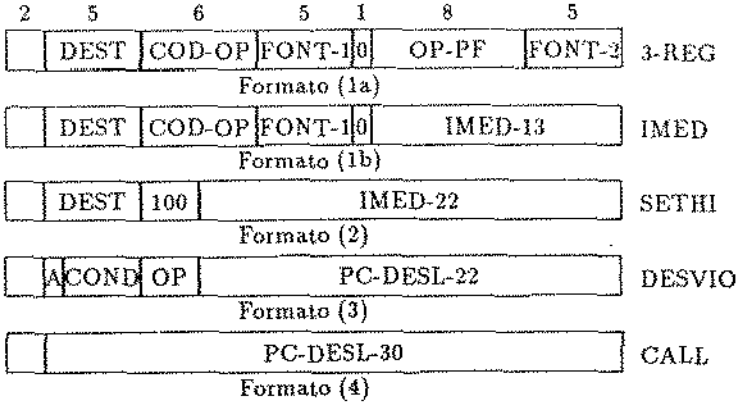


Figura 3.2: Formatos de Instruções na SPARC.

Variações das instruções de adição e subtração são suportadas para aumentar a eficiência do código gerado para linguagens com verificação de tipos dinâmica. Nestas situações, a operação é restrita aos 30 *bits* mais significativos e antes de aplicá-la a UI verifica se os dois *bits* de baixa ordem de um operando são iguais ao do outro (mesmo tipo).

Operações sobre constantes maiores de 13 *bits* tomam três instruções: `sethi` (formato 2) move os 22 *bits* mais altos da constante para um registrador, a aplicação da instrução de adição (`add`) sobre este registrador e os 13 *bits* mais baixos deixa a constante em registrador, finalmente uma instrução entre dois registradores executa a operação desejada.[1]

[1] Em função desta assimetria na arquitetura, compiladores reservam um registrador para o armazenamento tem-

As instruções LOAD/STORE seguem os formatos (1a) e (1b). Há instruções operando sobre *bytes*, meias-palavras, palavras ou duplas-palavras. O endereçamento da memória é *big endian* ou seja, dentro de uma palavra os *bytes* tornam-se menos significativos à medida que o endereço cresce.

Dois modos de endereçamento são diretamente suportados: registrador + registrador e registrador + deslocamento. Em particular, se $r0$ é empregado, mais dois modos são sintetizados: indireto por registrador e imediato. No VAX estes quatro modos são aplicados em 92% dos endereçamentos [Muc90a]. Deslocamentos de 13 *bits* são geralmente suficientes para acesso a variáveis locais, de acordo com [Dit82].

Instruções de desvio (b_{cc}) testam uma condição (eventualmente sempre válida) e somam ao contador de programa um deslocamento (sinalizado) de 24 *bits*, conforme o formato (3) (observe-se que instruções são alinhadas em múltiplos de quatro *bytes*). A instrução seguindo o desvio pode ou não ser executada, o controle é feito pela resolução da condição e pelo *bit* A (formato 3), detalhes na seção 3.5. A instrução desvia-e-liga (`jmpl`) têm comportamento semelhante, mas o endereço é calculado a partir de um dos modos de endereçamento mencionados anteriormente.

Em chamadas de procedimento (`call`) o deslocamento é de 32 *bits* (formato 4) e a instrução seguinte é sempre executada. O endereço de retorno é colocado no registrador ($r15$). O retorno é sintetizado através da instrução desvia-e-liga.

A janela ativa é avançada pela instrução `save`, normalmente a primeira executada em um procedimento. A instrução `restore` retrocede a janela corrente.

Instruções da UPF têm um único código de entrada na UI. O campo OP-PF no formato (1a) especifica a operação na UPF. Operações na UPF incluem adição, subtração, multiplicação, divisão e raiz quadrada. Precisão simples, dupla ou estendida é disponível em todas as situações, de acordo com a especificação IEEE 754-85 [Tan90].

3.3 Interação entre *hardware* e *software*

O ambiente de execução dos programas na SPARC é resultado da interação entre *hardware* e *software*. Em alguns casos a arquitetura reserva recursos a propósitos específicos, em outros é o sistema operacional que o faz. O compilador deve estar informado das convenções para gerar código correto. Esta seção detalha as implicações de convenções de *hardware* e *software* no processo de compilação.

3.3.1 Convenções de *Software*

Durante a execução de um procedimento, os registradores de uma janela são empregados para três finalidades: receber parâmetros, alocação a variáveis e passagem de parâmetros a outros procedimentos. A SPARC não tem registradores ou instruções explícitas sobre pilhas, de modo que o gerenciamento da pilha de execução de um programa é resultado das seguintes convenções de *software*:

1. na criação de um processo o sistema operacional reserva uma área para a pilha. Em qualquer instante, o registrador $r14$ deverá apontar para o topo da pilha (sp) e o registrador $r30$ (fp) para o início do registro de ativação corrente.

porário da constante. GCC reserva r1.

2. a pilha cresce dos endereços maiores para os menores, sempre alinhada em múltiplos de oito *bytes*;

3. os registradores $r16$ a $r31$ são preservados entre avanços e retrocessos de janelas, os demais podem ser corrompidos;

4. sempre que uma nova janela é estabelecida (via instrução `save`), espaço no topo na pilha deve ser reservado para armazenar os registradores desta janela em caso de estouro do conjunto de registradores;

5. a área reservada para tratar estouros deve iniciar na posição indicada por *sp*, mesmo quando a altura da pilha varia (por exemplo, através de alocação dinâmica de memória). Sua extensão é de 64 bytes ($[sp]$-$[sp+63]$). O alinhamento em múltiplos de oito *bytes* é necessário porque os tratadores de estouros pressupõem esta característica.

A arquitetura reserva mais dois registradores para guardar endereços de retorno. Assim o número de parâmetros passados (recebidos) em registradores é restrito a seis. A partir do sétimo, a pilha é usada. Um registro de ativação de um procedimento imediatamente antes de um nova chamada é mostrado na Figura 3.3. Espaço na pilha é reservado para tratar estouro, armazenar todos os parâmetros de saída[2], manter variáveis locais em memória e gerenciar registradores de ponto flutuante possivelmente necessários após a chamada.

Decorre da quinta convenção que os parâmetros de um procedimento são encontrados a partir do endereço $fp+64$, as variáveis locais a partir de fp e os parâmetros para chamadas a outros procedimentos a partir de $sp+64$. Pela primeira, a cadeia dinâmica [Kow83] é gerada automaticamente quando o procedimento avança a janela.

3.3.2 Gerenciamento de Janelas

Na visão do compilador o controle da janela ativa é feito pelas instruções `save` e `restore`. O controle interno usa mais dois registradores: CWP (*Current Window Pointer*) que indica a janela corrente e WIM (*Window Invalid Mask*) – um *bit* para cada janela implementada, usado para detetar estouros. Na discussão que se segue supõe-se N o número de janelas implementadas.

Enquanto há janelas disponíveis o funcionamento de `save` é similar a uma adição, à exceção que os operandos são buscados na janela ativa antes da instrução e o resultado armazenado na nova janela. Adicionalmente o CWP é decrementado (módulo N) para "indexar" a "próxima" janela. Esta semântica faz possível a `save` alocar espaço para o novo registro de ativação e atualizar o índice de janelas. Por exemplo, `save` $sp, -96, sp$ "cria" um novo registro de ativação com o *sp* da nova janela deslocado 96 bytes para cima na pilha e o fp assumindo o valor do velho *sp*, dada a correspondência entre o *sp* de uma janela e o fp da seguinte.

A instrução `restore` tem o comportamento simétrico: incrementa o CWP (módulo N) e faz uma adição entre operandos na janela atual e armazena o resultado num registrador da janela para onde o CWP passará a apontar. Note-se, entretanto, que o simples retrocesso de uma janela já restabelece a cadeia dinâmica, assim, normalmente não há funcionalidade para a adição do `restore`, sendo o caso comum: `restore` $r0, r0, r0$.

[2]Mesmo que alguns deles tenham seus valores em registradores e não em memória. Assim se o procedimento chamado necessitar armazená-los, já há espaço alocado.

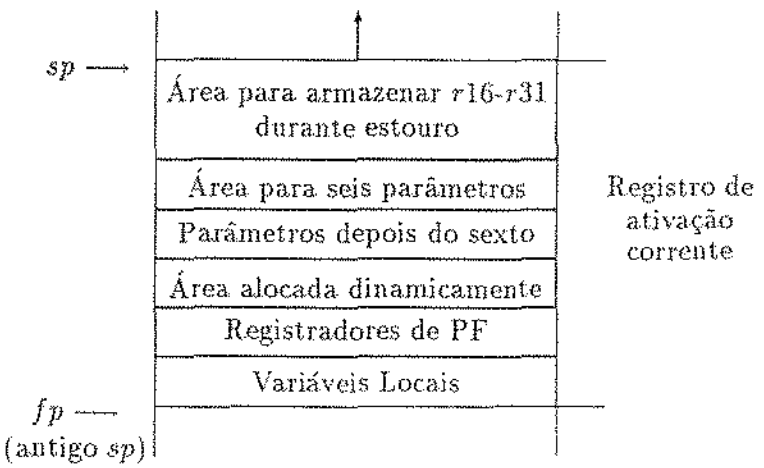


Figura 3.3: Pilha de execução de um programa.

Uma cadeia de N, ou mais, chamadas (ou retornos) sucessivas gera um estouro, ou seja, não há mais janela desocupada para associar ao procedimento chamado. O algoritmo que deteta e gerencia estouros é bastante engenhoso. Baseia-se na seguinte observação: em um conjunto circular, uma das janelas não pode ser utilizada totalmente, caso contrário, os registradores das vizinhas poderiam ser violados. Seja JT esta janela.

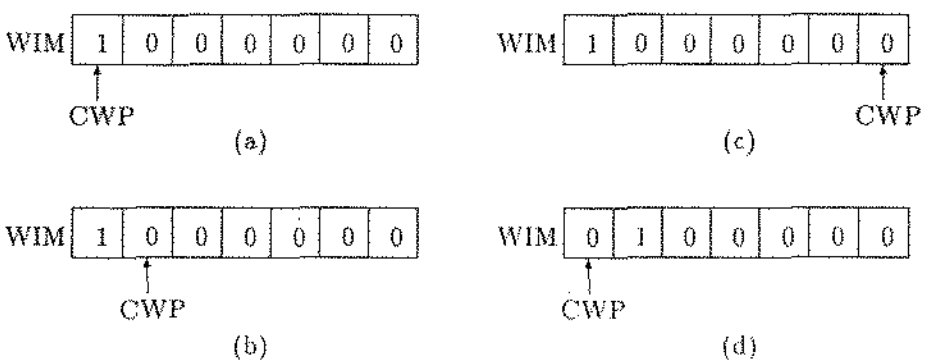


Figura 3.4: Gerenciamento do mecanismo de janelas deslizantes.

Inicialmente, um único *bit* em WIM é colocado para um pelo sistema operacional, no instante de criação do processo, indicando a JT. CWP também aponta para JT. Vide Figura 3.4(a), onde o valor de N é sete e JT corresponde à janela ativa. Depois da primeira execução de `save`, a Figura 3.4(b) mostra o novo valor de CWP. Imediatamente antes da *N-ésima* chamada, nenhum estouro é

detetado e o estado dos registradores que controlam o gerenciamento de janela é mostrado na parte (c) da mesma Figura.

Quando a instrução `save` decrementa o CWP, é necessário verificar se a nova janela coincide com JT. Se isto acontece, estouro foi detetado e uma rotina do sistema operacional é invocada. O tratador faz uma rotação simples para a direita em WIM e o *bit* em um indica a janela a armazenar. Pela quinta convenção é possível saber onde armazená-la. Em resumo, o procedimento que chamou `save` recebe JT e este passa a apontar para a janela liberada. O novo estado dos registradores WIM e CWP é mostrado na Figura 3.4(d).

Esquematicamente, a instrução `save`:

- Obtém os operandos em CWP;
- Decrementa CWP (módulo N);
- Verifica se CWP atingiu JT ($2^{CWP} = WIM$), neste caso:
  - Rotaciona WIM para a direita;
  - Armazena a janela $\log_2 WIM$ (que passa a ser JT);
- Soma os operandos;
- Armazena os resultados no registrador alvo (em CWP);

Se `restore` deteta estouro (mecanismo igual ao de `save`), o conteúdo original da janela apontada por JT deve ser recuperado da memória. O registrador *sp* na própria JT indica de onde os dados podem ser encontrados. É possível também ver que se a chamada de um procedimento causa estouro, o seu retorno não necessariamente causará, pois janelas são restauradas sob demanda.

As convenções são muito importantes para facilitar o gerenciamento de estouros, em especial a quinta. Considerando que o estabelecimento cadeia dinâmica e a alocação da área para o registro de ativação são automaticamente executados pela própria instrução `save`, avançar a janela corrente não acarreta, no caso básico, qualquer ineficiência em relação a um único arquivo de registradores, pelo menos em número de ciclos. O mesmo vale para `restore`.

JT também tem outra finalidade: tratamento de interrupções e *traps*. O sistema operacional deve ter uma janela disponível a qualquer momento para executar, por exemplo, o código de tratadores de interrupções. Obviamente o uso de JT é restrito aos registradores não compartilhados com outras janelas ($r16$-$r23$).

3.4 Implementação da Arquitetura

A arquitetura não define as organizações de pipeline ou *cache* a serem seguidas por implementações. Os compiladores podem, contudo, tirar vantagem da implementação e evitar perda de ciclos com *interlocks* em *hardware*. Esta seção justifica em termos de implementação da arquitetura os aspectos envolvidos com reorganização de código.

Implementações SPARC [Irl91, Sun87] têm adotado o seguinte esquema de *pipeline*:

Busca (B): a UI busca uma nova instrução neste estágio;

Decodificação (D): a instrução buscada no ciclo anterior é decodificada e seus operandos "lidos" do arquivo de registradores;

Execução (E): Execução da instrução buscada no ciclo anterior e armazenamento dos resultados em registradores temporários da UI;

Gravação (G): Escrita dos resultados no arquivo de registradores.

Existe um curto-circuito entre os estágios (E) e (D), de maneira que não há atraso entre duas instruções cujos dados acabam de ser calculados no estágio (E), mas ainda não foram armazenados no arquivo de registradores.

3.4.1 Custo e Latência das Instruções

A grande maioria das instruções, tipicamente as aritméticas e lógicas, tem custo de apenas um ciclo e nenhuma latência.

O *cache* e o barramento compartilhados (Figura 3.1) não permitem a busca simultânea de instruções e dados. Portanto, instruções que fazem acesso à memória custam, no mínimo, dois ciclos. A instrução STORE custa mais um ciclo (total de três) porque as implementações só possuem duas portas de leitura ao arquivo de registradores e a instrução necessita buscar três operandos. A instrução LOAD tem latência de um ciclo. A razão para isto é que o curto-circuito entre os estágios (E) e (D) do *pipeline* não pode ser usado neste caso, pois (E) calcula o endereço do operando e não o operando em si.

Desvios condicionais tomados custam apenas um ciclo. Se a condição é falsa o custo é aumentado em um ciclo. Desvios tomados são mais "baratos" porque o *hardware* pressupõe o desvio. Enquanto a instrução de desvio está no estágio (E), já existem duas outras no *pipeline*. No estágio (D) está a que segue o desvio, no (B) está a instrução alvo do desvio. Se o desvio não acontece, a busca é repetida, aumentando o custo. Igualmente, a anulação da instrução seguinte ao desvio aumenta o custo do desvio por um ciclo. Portanto, no pior dos casos, uma instrução de desvio pode custar até três ciclos.

Instruções da UPF tomam dois ciclos na UI. Durante o estágio (E) a instrução é enviada à UPF. No estágio seguinte (G), o endereço da instrução é enviado; uma nova instrução não é executada neste ciclo para não corromper o endereço a ser enviado. Na UPF a instrução toma ciclos adicionais, mas operando paralelamente à UI. Os custos e latências na UPF não nos são disponíveis, mas sua interface com a UI permite, inclusive que seja não *pipelined*.

A Tabela 3.1 resume os custos e latências das instruções na UI.

3.4.2 O *cache*

As implementações atuais possuem um *cache* de 64 Kb, organizado em mapeamento direto e com linhas de 64 bytes. A política de atualização *cache*/memória é *write-through*.

Irlan [Irl91] determinou empiricamente o custo de acesso a dados fora do *cache* na SPARCstation 1. Seus resultados implicam em um baixo desempenho do sub-sistema de memória destas máquinas. Do ponto de vista de reorganização de código, sua principal observação é que mesmo um STORE de 64 *bits* é suficiente para colocar a UI em estado de espera, em função dos poucos *buffers* de escrita na memória. Por isto, recomenda que quaisquer seqüências de instruções STORE de até 32

INSTRUÇÕES	CUSTO	LATÊNCIA	OBSERVAÇÕES
LOAD (até 32 *bits*)	2	1	
LOAD (64 *bits*)	3	1	
STORE (até 32 *bits*)	3	-	
STORE (64 *bits*)	4	-	
DESVIO (condicional)	1	-	+1 no custo, se não tomado
		-	+1 no custo, se prox. inst. anulada
DESVIO (incondicional)	1	-	+1 no custo, se prox. inst. anulada
DESVIA-E-LIGA	2	-	Prox. inst. sempre executada
CALL	1	-	Prox. inst. sempre executada
PF-ops	2	-	Têm custo (latência) extra na UPF
Demais	1	-	

Tabela 3.1: Custo e Latência das instruções na UI.

bits sejam separadas por três ou quatro (ele não chegou a concluir precisamente) outras que operam sobre registradores ou constantes. No caso de STORE de 64 *bits*, o custo da instrução é estimado em oito ciclos, dada a impossibilidade de evitar o atraso entre os dois armazenamentos de 32 *bits* que compõem a instrução.

3.4.3 A UPF

A UPF é composta por duas unidades de execução que operam em paralelo: um somador e um multiplicador de ponto flutuante. Uma unidade de controle despacha instruções para ambas as unidades de execução à medida que as recebe da UI. A UPF tem ainda um descritor para cada uma das instruções em execução.

Quando a UPF recebe uma instrução da UI, checa dependências de recursos e operandos da instrução recebida. Se dependências existem, a UI é bloqueada até que a dependência deixe de existir. Quando não existem dependências, no ciclo seguinte a UI envia o endereço da instrução. O endereço é mantido na UPF para que a UI não necessite manter contexto algum relativo a operações de ponto flutuante. Se alguma exceção acontece, a UPF tem toda a informação para reiniciar instruções pendentes.

Há dependência de recursos quando a unidade de controle não pode despachar a instrução recebida para a unidade de execução adequada, ou porque ambas estão ocupadas, ou existe uma livre, mas do tipo "errado". Desde que a UI pode despachar uma nova instrução para a UPF a cada dois ciclos, um balanceamento de desempenho é obtido se a UPF também aceita instruções nesta taxa.

Dependências de operandos acontecem quando a instrução requer um resultado ainda não disponível; estas são amenizadas através de curto-circuito.

Para operações de LOAD/STORE, UI e UPF operam em conjunto. A UI gera o endereço e envia sinais de controle à UPF para esta colocar ou receber o dado no barramento. A UPF bloqueia a UI se o dado não está disponível, no caso de armazenamento, ou ainda será utilizado, no caso de leitura.

SPARC também suporta desvio em função de códigos de condição da UPF. O registrador de

código de condição da UPF é replicado na UI. Sempre que a UPF recebe uma instrução que modifica os seus códigos de condição, a informação da UI é invalidada e posteriormente atualizada. O tempo de invalidação deste dado requer que a instrução de comparação e o desvio sobre os códigos de condição da UPF sejam separadas por uma instrução da UI.

3.5 Reorganização de Código na SPARC

A SPARC oferece um mecanismo que pode ser útil durante o processo de reorganização de código: o *bit* de anulação para instruções seguindo desvios. Seu efeito é mostrado na Tabela 3.2.

BIT A	TIPO DE DESVIO	EXEC. PROX.?
a = 0	Incondicional	Sim
	Condicional(tomado)	Sim
	Condicional(não tomado)	Sim
a = 1	Incondicional	Não
	Condicional(tomado)	Sim
	Condicional(não tomado)	Não

Tabela 3.2: Execução da instrução seguindo um desvio.

Geração de código para construções estruturadas pode utilizar o *bit* A efetivamente em construções repete-até (Figura 3.5) e se-então-senão (Figure 3.6). O código na Figura 3.5 baseia-se no fato de que malhas[3] são, em geral, executadas mais de uma vez, sendo, portanto, simples e conveniente aproveitar o *slot* que sucede o desvio com a instrução para a qual o controle é normalmente passado. O alvo do desvio é mudado de acordo e a anulação garante que a semântica do programa não é alterada caso o desvio não seja tomado.

SEM REORGANIZAÇÃO		COM REORGANIZAÇÃO	
END	INSTRUÇÃO	END	INSTRUÇÃO
	add ...		add ...
L:	sub ...	L:	sub ...
	or ...	L':	or ...
	:		:
	:		:
	b_{cc},a=0 L		b_{cc},a=1 L'
	nop		sub ...

Figura 3.5: Reorganização em construções repete-até.

A boa utilização do *bit* de anulação em construções condicionais é mais problemática em função do comportamento menos previsível do desvio. De posse de alguma informação, por exemplo *profile*, alguma das alternativas na Figura 3.6 pode ser empregada. Na parte (a) o SENAO é supostamente

[3]Compiladores otimizadores normalmente convertem construções enquanto-faça para repete-até porque estas executam uma instrução de desvio a menos para cada iteração.

mais freqüentemente executado, na (b) o ENTAO. Desviar para ENTAO ou SENAO requer inversão na condição que gerencia o desvio. Isto não é problema porque a SPARC oferece para cada possível teste de condição a sua forma negativa.

No caso de desvios incondicionais a anulação tem o objetivo de reduzir o tamanho do código: se nenhuma instrução dentro do bloco básico onde está o desvio pode ser encontrada para colocar no *slot*, o *bit* de anulação serve para evitar a inserção explícita do nop.

SEM REORGANIZAÇÃO		COM REORGANIZAÇÃO (A)		COM REORGANIZAÇÃO (B)	
END	INSTRUÇÃO	END	INSTRUÇÃO	END	INSTRUÇÃO
	b_{cc},a=0 SENAO		b_{cc},a=1 SENAO		$b_{cc'}$,a=1 ENTAO
	nop ...		add ...		sub ...
ENTAO:	sub ...	ENTAO:	sub ...	SENAO:	add ...
	xor ...		xor ...		or ...
	sethi ...		sethi ...		and ...
	⋮		⋮		⋮
	GO ...		GO ...		GO ...
SENAO:	add ...	SENAO:	or ...	ENTAO:	xor ...
	or ...		and ...		sethi ...
	and ...		⋮		⋮
	⋮		⋮		⋮

Figura 3.6: Reorganização em construções se-então-senão.

Instruções call e jmpl e desvios incondicionais (b) têm um tratamento elementar. Na grande maioria dos casos, é suficiente inverter a ordem de uma destas instruções com a que a precede. Exceções são blocos básicos com uma única instrução.

Algoritmos mais elaborados, descritos no capítulo 6, tratam os seguintes casos:

- LOAD seguido de uso do registrador carregado: separação por uma instrução qualquer;
- comparação e desvio sobre código de condição da UPF: separação por uma instrução da UI;
- STORE seguido de STORE: separação por três outras instruções quaisquer.

Dependendo da UPF pode ser fortemente recomendável separar instruções de ponto flutuante por uma ou mais inteiras.

3.6 Conclusões

A geração de código SPARC para linguagens com encaixamento estático requer atenção com relação ao ambiente não local aos procedimentos. Variáveis locais de um procedimento P, referenciadas fora de P, não podem estar em registradores quando P faz novas chamadas, caso contrário o esquema de janelas pode deixar o dado inacessível. Felizmente é possível conhecer todas as variáveis com este comportamento antes de fazer a alocação de registradores em P e mantê-las, pelo menos temporariamente, em memória.

Em linguagens como C, os registradores globais podem ser empregados para acesso a 2^{13} *bytes* do ambiente global por registrador. Em linguagens como Pascal, eles podem ser utilizados como registradores de base para os registros de ativação dos procedimentos [Kow83].

A arquitetura descrita neste capítulo é a versão 7. A próxima versão, já em projeto [Muc90a] deverá conter instruções de multiplicação e divisão embutidas no *hardware*. Estudos preliminares para implementações superescalares podem ser encontradas em [LKB91].

Capítulo 4

Alocação Intraprocedimental de Registradores

"Uma cabeça oca não está vazia. Na realidade, está cheia de asneiras. Daí a dificuldade em meter qualquer coisa dentro dela."
Eric Hoffer

4.1 Introdução

Na compilação de um programa envolvendo V variáveis para uma máquina alvo que dispõe de R_f registradores físicos ($V > R_f$) é necessário ter uma regra que designe quais variáveis devem estar nos registradores em determinado instante. É também desejável ter critérios para reduzir o número de acessos à memória decorrentes da movimentação de dados entre registradores e memória. Este problema é denominado alocação de registradores. A emissão de código que executa sobre operandos em registradores pode ser uma necessidade, caso de máquinas LOAD/STORE, ou motivada pela eficiência. Sabe-se dos capítulos precedentes que operandos em registradores diminuem o tempo de execução das instruções.

Alocação de registradores não somente é uma grande fonte de otimização como também afeta os resultados de outras técnicas. Por exemplo, a eliminação de subexpressões comuns [ASU86] gera um conjunto de temporários onde os resultados de algumas computações podem ser encontrados; se o alocador não for capaz de fazer associações adequadas destes temporários a registradores, então recomputar expressões pode ser menos oneroso do que utilizar os temporários, tornando negativos os efeitos daquela otimização.

Em arquiteturas LOAD/STORE alocação é particularmente importante e motivada porque:

- O custo (relativo) de acesso à memória é alto;
- A ausência de alocação implica em buscar a variável da memória antes de cada uso e armazená-la após cada definição;
- Existe, normalmente, um grande conjunto de registradores disponível.

O objetivo final é encontrar a melhor maneira de associar variáveis a registradores de tal modo que o tempo de execução do programa objeto seja minimizado. Os algoritmos para resolver este

problema variam em função do intervalo de programa sobre o qual fazem a alocação. Exemplos de possíveis intervalos são blocos básicos, malhas, procedimentos e programas. À medida que a granularidade do intervalo aumenta, o código emitido tende a ser globalmente mais eficiente, no entanto, o custo para obtê-lo cresce muito mais rapidamente que a qualidade dos resultados. O processo de alocação é normalmente dividido em três partes:

- Determinação das variáveis que podem compartilhar um mesmo registrador;
- Estimativa das variáveis mais adequadas para residirem em registradores;
- Associação entre variáveis e registradores.

Este capítulo trata de algoritmos cuja unidade de alocação é menor do que ou igual a um procedimento; técnicas para alocação global a um programa são descritas no capítulo 5. A seção seguinte trata de alocação dentro de blocos básicos, na próxima um processo é sugerido para malhas. A seção 4.4 trata de alocação para um procedimento inteiro. Passagem de parâmetros em registradores e algumas convenções úteis em otimização e as respectivas interações com o alocador são descritos na seção 4.5. A seção 4.6 conclui.

Terminologia

A interação entre o gerador de código e o alocador de registradores pode acontecer de dois modos. No primeiro, o gerador supõe a inexistência do alocador e coloca todas as variáveis em memória, usando uns poucos registradores na emissão do código. O trabalho do alocador é escolher algumas variáveis e promovê-las a registradores. Alternativamente, o gerador de código pode supor a existência de infinitos registradores simbólicos, associando um a cada variável escalar do programa fonte ou temporário gerado ao longo do processo de otimização. Cabe ao alocador mapear registradores simbólicos a registradores físicos. Não é feita distinção entre os casos porque conceitualmente são equivalentes.

Uma variável é derramada[1] se foi escolhida para ficar na memória.

Um *ponto de execução* refere-se a um instante entre a execução de duas instruções sucessivas. Há um *caminho* da instrução i até a instrução j se existe alguma possível execução onde ambas as instruções são executadas, tal que uma execução de i acontece antes de uma execução de j.

Um *uso* da variável v na instrução i indica que o valor de v é um operando fonte da instrução i. Analogamente, uma *definição* da variável v na instrução i indica que v passa a ter o resultado da computação efetuada em i. *Referência* a uma variável v, $ref(v)$, diz respeito a uma definição ou um uso de v. A profundidade de uma referência a v, $prof(ref(v))$ é o nível de encaixamento da malha onde a referência acontece (zero, se fora de qualquer malha).

A freqüência de execução de um bloco básico b, $F(b)$, é o número estimado de vezes que ele será executado a cada chamada do procedimento ao qual pertence. A estimativa pode ser estática, por exemplo baseada no nível da malha onde o bloco ocorre, ou dinâmica, através de *profile*.

Uma variável v está viva em um ponto P_1 se existe algum caminho C, de P_1 até um outro ponto P_2, tal que P_2 esteja imediatamente antes de um uso de v e não existe definição (de v) ao longo de C. Também se diz que v *alcança* um ponto P_2 se existe algum caminho de um ponto P_1, imediatamente após uma referência a v, até P_2.

[1] do inglês: *spilled*

A *vida* de uma variável v é formada por todos os pontos P, tais que v alcança e está viva em P. Cada um dos componentes conexos da vida de v é denominado uma *cadeia* de v.

Neste capítulo, o termo *alocação global* refere-se à alocação que se espalha ao longo de um procedimento inteiro. Note-se que toda a informação acima definida é resultado da análise de fluxo de controle e de dados, normalmente executada em compiladores otimizadores.

4.2 Alocação em Blocos Básicos

Considerando uma máquina LOAD/STORE, onde o resultado da computação é atribuído a um dos operandos (instruções de dois endereços), Aho *et al* [ASU86] apresentam um algoritmo eficiente para geração de código ótimo para expressões. O número mínimo de registradores também é garantido, mas a abrangência é muito pequena: restringe-se a situações onde nenhum dos resultados intermediários pode ser reutilizado, em outras palavras, o algoritmo se aplica exclusivamente a árvores de avaliação de expressões.[2] Se reutilização de operandos é permitida, o que pode ser visto como a modelagem de um bloco básico, Aho *et al* [AJU77] provaram que a geração de código ótimo (para a máquina em questão) é um problema *NP-completo*, mesmo quando o número de registradores disponível é infinito. A complexidade resulta da dificuldade em determinar a ordem de avaliação da expressão, visto que em máquinas com instruções de dois endereços, um dos operandos é destruído em cada operação. A escolha de uma ordem de avaliação, que não é ótima, implica em usar mais instruções e registradores para manter valores temporários que são destruídos pelas instruções.

Com um modelo capaz de armazenar resultados em um terceiro operando e infinitos registradores, a geração de código ótimo é muito simples e consiste basicamente em avaliar o grafo que representa a expressão de maneira ascendente, nível a nível, associando um registrador distinto a cada resultado intermediário.

Em condições reais, o número de registradores é sempre finito. Fixada uma ordem de avaliação, a geração de código ótimo passa a depender do número de registradores disponíveis, mas o problema de determinar o número mínimo que garante esta otimalidade, R_b, é facilmente solucionável. Eventualmente uma outra ordem de avaliação pode necessitar de menos registradores. A minimização requer que dentro do bloco básico uma variável seja carregada em registrador no ponto que antecede o primeiro uso. Da mesma forma, qualquer variável (re)definida dentro do bloco deve ser armazenada no ponto seguinte ao último uso. Dadas estas restrições, R_b é exatamente o maior número de variáveis vivas em qualquer dos pontos do bloco básico. Na Figura 4.1, quatro registradores são necessários, determinados pelas variáveis vivas antes da segunda instrução.

Quando R_f é menor que R_b, derramamento de variáveis é necessário. Um algoritmo genérico e ótimo para minimizar derramamento também é exponencial. Observe que em cada ponto do bloco básico existe um conjunto de configurações possíveis para mapear variáveis a registradores. Entre configurações relacionadas a dois pontos consecutivos existe um custo de passar de uma para outra, proveniente das possíveis movimentações entre registradores e memória. Determinar o menor custo de derramamento é calcular o menor caminho entre uma configuração relativa à primeira instrução e uma relativa à última. Existe um conjunto de técnicas para reduzir o número de configurações associadas a cada instrução, para detalhes veja [HKMW66].

[2] A questão da seleção de código também é ignorada, supõe-se que cada operação tem uma instrução de máquina que a mapeia diretamente.

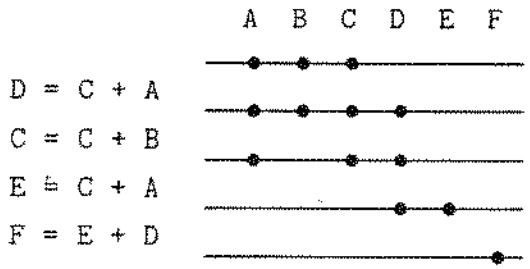


Figura 4.1: Variáveis vivas em um trecho seqüencial de programa.

Uma estratégia mais simples consiste em determinar o lucro de alocar uma variável, priorizando a alocação das variáveis mais referenciadas; variáveis que não estão vivas simultaneamente em qualquer ponto do bloco básico podem usar os mesmos registradores. Alguns registradores devem ser dedicados a gerenciar derramamento.

Sumarizando: a geração de código ótimo é computacionalmente difícil, mesmo com infinitos registradores, se a máquina tem instruções de dois endereços. Se esta condição é relaxada, o problema se torna muito simples. Em termos práticos o número finito de registradores deixa o problema, de novo, difícil quando derramamento deve ser gerado. Adicionalmente, a alteração da ordem das instruções poderia modificar a necessidade de derramamento.

4.3 Alocação em Malhas

Alocação restrita a blocos básicos gera código ineficiente. Todas as variáveis não locais ao bloco necessitam ser carregadas antes do primeiro uso e armazenadas após a última referência, se redefinidas dentro (e vivas ao final) do bloco.

Desde que procedimentos, em geral, gastam grande parte do tempo executando código dentro de malhas pode ser útil estender a visão do alocador a estas regiões. A forma mais simples de estender a alocação é calcular o lucro ao longo de todos os blocos componentes da malha, escolhendo as variáveis mais utilizadas. O custo de armazenar (restaurar) as variáveis em memória fica restrito aos pontos de entrada e saída da região.

A alocação poderia iniciar nas malhas mais internas (desde que supõe-se serem as mais freqüentemente executadas) e progressivamente atingir os níveis mais externos. Neste processo, há vantagens de preservar as associações dos níveis mais internos. Por exemplo, suponha duas malhas, M_1 e M_2, tal que M_1 está (encaixada) dentro de M_2. Se a mesma variável é associada a registradores diferentes em M_1 e nos demais pontos de M_2, é necessário introduzir operações de cópia entre os registradores. Métodos de priorizar este tipo de associação são discutidos na seção 4.5.

4.4 Alocação Global

A alocação global de registradores é normalmente formulada como um problema de coloração de grafos. Neste método, cada vértice no grafo representa um candidato a alocação, tipicamente uma variável e um conjunto de pontos pertencentes à sua vida. Existe uma aresta entre dois destes vértices se em algum ponto do procedimento os candidatos por eles representados estão vivos simultaneamente. Este grafo é denominado *grafo de interferências ou conflitos*. Uma aresta representa o *conflito*.

O problema da coloração é associar um conjunto fixo de cores aos vértices do grafo garantindo que vértices adjacentes recebam cores distintas. Intuitivamente isto representa o fato de que duas variáveis vivas em um mesmo ponto de execução não podem compartilhar o mesmo registrador. Um grafo é *k-colorável* se ele pode ser colorido com k cores. O número mínimo de cores para colorir um grafo é chamado *número cromático*. Em um grafo arbitrário, encontrar uma *k-coloração* é um problema *NP-completo* ($k \geq 2$) [Man89]. O grau de um vértice v, $grau(v)$, é o número de vizinhos de v no grafo. Os termos candidato e vértice e registrador e cor estarão sendo usados de maneira intercambiável.

Independentemente das adversidades, o algoritmo de alocação deve encontrar uma coloração para o grafo. Este processo pode implicar na mudança dos candidatos a serem coloridos. É necessário, portanto, estabelecer um critério para coloração que gere associações válidas, sem, entretanto, exigir a otimalidade. A principal questão envolvida com algoritmos de alocação via coloração é derramamento. Ao contrário de alocação dentro de blocos básicos, o número mínimo de registradores para colorir o grafo não pode ser determinado de modo eficiente.[3] Ou seja, pode acontecer que mesmo dispondo de registradores suficientes, código de derramamento seja emitido. É possível mostrar também que qualquer grafo pode ser resultante de interferências entre variáveis de um procedimento (mesmo se somente construções estruturadas são usadas), ou seja, que alocação de registradores em um procedimento é realmente *NP-completo*.

Portanto, resta o uso de funções heurísticas para definir as variáveis a serem promovidas para registradores. Novamente, o principal guia da função heurística é o número de referências à variável, neste caso, ao longo do procedimento.

Nas próximas subseções são descritos três métodos de alocação global. Todos usam, em algum estágio, coloração de grafos. Os dois primeiros são importantes por serem empregados na maioria dos compiladores disponíveis comercialmente. O último é a formalização de alocação em regiões: a alocação inicia em pequenas regiões e se expande até atingir a totalidade do procedimento, mantendo preferencialmente associações prévias. Coloração é usada nos vários estágios.

4.4.1 Alocação por Coloração de Grafos

Embora a idéia de alocar registradores via coloração de grafos seja antiga, a primeira implementação ocorreu em 1981, como apresentada em [CAC+81] e posteriormente em [Cha82a]. A principal característica desta implementação é que ela é sistemática e uniforme. Nenhum esforço de alocação local, através de um conjunto específico de registradores é necessário, o alocador é capaz de identificar a alocação dentro de blocos básicos como um caso particular de alocação global.

[3] Mesmo já havendo sido feita alguma escolha sobre ordem de avaliação de expressões.

O Grafo de Interferências

O mapeamento entre variáveis e candidatos é muitos-para-muitos e não um-para-um, dado que duas cadeias disjuntas podem receber registradores distintos, mesmo que se refiram à mesma variável (definida pelo programador). Gerar dois candidatos nestas situações é importante porque dá mais flexibilidade ao alocador para a geração de uma R_f-coloração. Algumas destas cadeias também podem ser combinadas, como mostrado adiante, de modo a forçar algumas associações que melhoram a qualidade do código final.

O conceito de interferência utilizado é ligeiramente diferente, mas igualmente expressivo: dois candidatos interferem se em um ponto imediatamente seguindo uma definição de um deles o outro está vivo e pode ter um valor diferente do primeiro. Este conceito tem a vantagem de gerar menos interferências e, em alguns casos, diminuir o número cromático, facilitando a coloração.

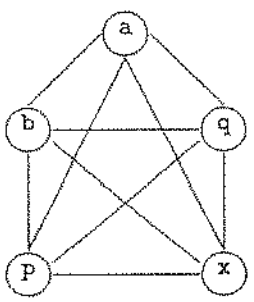
Por exemplo, suponha o procedimento (função) escrito em linguagem C, mostrado na Figura 4.2(a), onde parâmetros não são tidos como candidatos a alocação. O cálculo de interferências usando o método aqui apresentado conduz ao grafo da Figura 4.3(a) que pode ser colorido com três cores. Em se considerando candidatos vivos simultaneamente em qualquer ponto como conflitantes, o grafo de interferências está na Figura 4.2(b) e tem número cromático igual a cinco.

```
void f(p1, p2)
int p1, p2;
{
   int a, b, x, p, q;

   p = p1;
   q = p2;
   if (p > q)
      a = f(p);
   else
      b = g(q);

   x = p;
   if (p > q)
      return(a + q);
   else
      return(a + x);
}
```

(a)



(b)

Figura 4.2: (a) Uma função escrita em Linguagem C e (b) grafo de interferências dos candidatos a alocação de registradores (método convencional).

O grafo da Figura 4.3(a) ainda mostra que as variáveis p e x podem efetivamente compartilhar um mesmo registrador, caso onde a cópia entre eles poderia ser removida. A generalização da situação é a seguinte: se existem cópias entre dois candidatos do grafo de interferências, c_i e c_j, e eles não interferem[4], então os candidatos podem ser combinados em um novo, c_k. Com isto c_i e c_j são retirados do grafo e todas as suas arestas passam a ser arestas de c_k.

Esta otimização, que Chaitin denominou de *subsumption*, pode aumentar o número cromático do grafo de interferências ou "dificultar" a sua coloração, porém é útil empregá-la porque existem

[4] Pela definição de interferência, cópias não geram conflitos.

PROGRAMA	Registradores inteiros	Registradores de de ponto flutuante
001.gcc-1.35	6,35	-
008.espresso	9,97	-
022.li	4,31	-
023.eqntott	7,37	-
navega	12,20	12,20
reorg	8,64	-
stanford	6,04	-

Tabela 4.1: Número médio de registradores (de uso geral) necessários por procedimento.

muitas cópias entre variáveis no momento da alocação. Uma fonte de tais instruções é o estabelecimento de convenções para passagem de parâmetros em registradores. O grafo da Figura 4.3(b) é o resultado de aplicar *subsumption* ao da Figura 4.3(a). Na seção 4.5 estes temas serão abordados em mais detalhes.

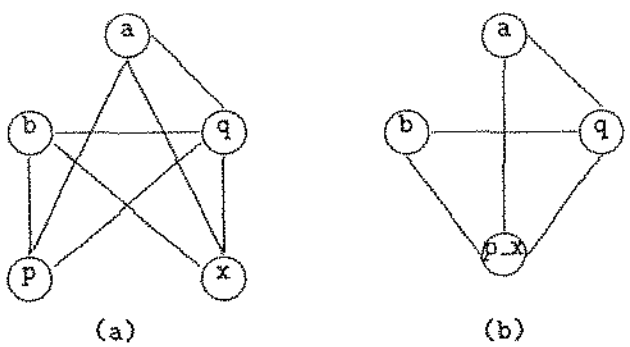


Figura 4.3: O grafo de interferências referente ao código intermediário da Figura 4.2 de acordo com a sugestão de Chaitin: sem (a) e com *subsumption* (b).

Coloração

Existem várias maneiras de colorir um grafo. Chaitin observou que na disponibilidade de 17 registradores de uso geral, quase sempre é possível conseguir rapidamente uma coloração através de um algoritmo que faz retrocesso e tem complexidade exponencial no pior caso. Dados coletados para vários *benchmarks* compilados pelo GCC[5], tendo como arquitetura alvo a SPARC, são apresentados na Tabela 4.1 e confirmam as observações de Chaitin. Uma abordagem mais simples, do ponto de vista do tempo de execução, divide o processo em dois estágios: simplificação e associação de cores.

Durante a simplificação a principal observação de Chaitin foi que um vértice com grau menor que R_f é colorável independentemente das cores dos seus vizinhos e, portanto, pode ser ignorado

[5]GCC aloca registradores empregando um método similar ao de Chaitin.

(juntamente com as suas arestas) em considerações posteriores a respeito da coloração. Desse ponto de vista, o grafo vai sendo reduzido até que se torna vazio.

Durante a simplificação, os vértices vão sendo rotulados com números crescentes à medida que vão sendo "eliminados". Cores são então associadas aos vértices, em ordem decrescente de rótulos, pois assim a condição que permitiu considerar cada um deles como "colorável" se mantem satisfeita.

O processo de associação dá oportunidade à coloração de vértices originalmente com grau maior que R_f, mas a coloração mínima não é garantida. Por exemplo, suponha o grafo simples da Figura 4.3(a). Três cores são suficientes para colori-lo (*a:verde, b:verde, p:azul, x:azul, q:vermelho*). O algoritmo acima, entretanto, necessita de, no mínimo quatro cores, caso contrário ocorrerá um bloqueio no processo de simplificação e algum candidato deverá ser removido do grafo e considerado como residente em memória.

Gerenciamento de Derramamento

Embora em casos de bloqueio do processo de simplificação, qualquer candidato possa ser escolhido para remoção do grafo de interferências, é conveniente selecionar aquele que minimiza o custo do código de derramamento. Chaitin sugere a seguinte função heurística para determinar o custo de derramar um candidato c:

$$custo(c) = \sum_{\forall ref(c)} 10^{prof(ref(c))}$$

Um alto custo de derramamento é atribuído a candidatos freqüentemente referenciadas dentro de malhas, estimulando o alocador a mantê-los em registradores. A decisão do candidato a ser derramado é suplementada com a informação do grau (envolvendo candidatos ainda não rotulados) de cada candidato no grafo. Quanto maior o grau de um vértice, maior é a probabilidade que ele permita a coloração de outros vértices, se removido. Portanto, quando um bloqueio acontece, o candidato c, a ser derramado é o que possui menor valor de $DERRAM$, onde:

$$DERRAM(c) = custo(c)/grau(c)$$

Existem ainda dois pontos importantes sobre derramamento. Em uma máquina LOAD/STORE e com todos os registradores disponíveis ao alocador, fazer o derramamento de um candidato não implica simplesmente em removê-lo do grafo de interferências, mas sim em substituí-lo por novos candidatos representando cada LOAD antes de um uso, ou STORE depois de uma definição. Assim, após fazer as decisões sobre candidatos a derramar, é necessário reconstruir o grafo e repetir todo o processo. Para desestimular posteriores derramamentos desses candidatos, suas funções de *custo* são pré-definidas com valores muito grandes.

A segunda observação é voltada para reduzir a quantidade de derramamento gerado. Suponha que uma variável derramada seja referenciada em dois pontos dentro de um mesmo bloco básico. Em geral, a variável deve ser substituída por dois outros candidatos representando os pontos onde as referências ocorrem. Se, no entanto, nenhuma outra variável morre entre as referências, é suficiente inserir apenas um candidato, pois a inserção de dois não facilita a coloração do grafo, em virtude deles gerarem conflitos com os mesmos candidatos. A inserção de apenas um, por outro lado, tem o efeito de diminuir a quantidade de derramamento. O mesmo raciocínio se aplica, obviamente, a qualquer número de referências sem morte de outras variáveis entre elas.

Sendo G um grafo de interferências, o algoritmo de Chaitin consiste nos passos mostrados na Figura 4.4, onde a função $grau(v)$ calcula as arestas envolvendo o vértice v e algum outro ainda não rotulado.

- Repita
  1. Considere todos os vértices como não rotulados;
  2. Inicie o rótulo corrente em zero;
  3. Enquanto existem vértices não rotulados:
     - Se existe um vértice não rotulado v, com $grau(v) < R_f$
       * atribua o rótulo corrente a v;
       * incremente o rótulo corrente de um;

       Senão
       * Escolha um vértice v, com menor $DERRAM(c)$;
       * Marque v como derramado;
       * remova v de G;
  4. Se nenhum vértice foi marcado como derramado
     - Associe cores a todos os vértices na ordem inversa à rotulação;

     Senão
     - Gere código de derramamento;
     - Reconstrua G;
- até que todos os vértices em G recebam cores;

Figura 4.4: Alocação por coloração de grafos.

Comentários

A uniformidade da técnica de coloração permite que várias situações encontradas em alocação de registradores sejam facilmente modeladas. Suponha, por exemplo, que em uma arquitetura particular a instrução de adição sempre armazena os resultados em um mesmo registrador. Inconsistências são evitadas simplesmente criando um vértice pré-associado àquele registrador e fazendo-o interferir com cada candidato que cruza uma instrução de adição.

Várias máquinas (vide apêndice A) têm o conjunto de registradores dividido em classes, normalmente, registradores inteiros e de ponto flutuante. Como as instruções trabalham sobre classes específicas, candidatos não podem ser associados arbitrariamente a qualquer registrador físico. Mesmo sendo possível construir um único grafo para associar todos os registradores, isto envolve mudança no conceito de interferência e aumenta consideravelmente o tamanho do grafo de interferências. A abordagem mais simples é dividir a alocação por classes, dando oportunidade, inclusive, à paralelização do processo, se a máquina onde o alocador executa assim permitir.

É comum também que os candidatos a alocação tenham necessidades diferentes em relação ao tamanho dos registradores, dependendo da precisão do tipo de dados que representam. Nas considerações a seguir, é suposto que quando registradores podem ser agrupados para tratar valores maiores, eles devem ser consecutivos, e cada registrador pertence a um único agrupamento de determinado tamanho. Esta suposição está de acordo com o adotado pela maioria das arquiteturas.

O número de registradores necessários para um candidato c é dado por $tam(c)$.

Durante a simplificação, um candidato pode ser rotulado se é garantido existir um registrador para ele durante o processo de associação; isto significa que o número de registradores usados pelos vizinhos deve ser inferior ao número disponível. Tome o caso de um candidato c de tamanho $tam(c)$ em um grafo de interferências onde candidatos têm tamanhos diferentes. É necessário definir precisamente dois números: a quantidade de agrupamentos do tamanho do candidato c, e o número deles que podem vir a ser usados pelos seus vizinhos. Diante das suposições, existem exatamente $\lfloor \frac{R_f}{tam(c)} \rfloor$ possíveis agrupamentos aos quais c pode ser associado. Note-se, entretanto, que cada um dos vizinhos de c pode potencialmente utilizar qualquer registrador e comprometer um agrupamento inteiro, no mínimo.

Generalizando estas idéias é possível determinar que um candidato pode ser rotulado durante a simplificação do grafo de interferências se:

$$\sum_{\forall v \in VIZ(c)} \left\lceil \frac{tam(v)}{tam(c)} \right\rceil < \left\lfloor \frac{R_f}{tam(c)} \right\rfloor$$

onde $VIZ(c)$ é o conjunto de vizinhos (atuais) do candidato c.

Observe-se que o tamanho também pode ser considerado na escolha do candidato a derramar. Como um candidato que ocupa muitos registradores pode dificultar mais o processo de coloração ele deve ser mais penalizado na função $DERRAM$. Assim o tamanho passa a ser mais um divisor de *custo*.

Alguns refinamentos foram propostos ao algoritmo básico. Bernstein *et al.* [BGM+90] sugere que a escolha dos candidatos a serem derramados seja gerada usando várias funções de heurística, o que equivale a executar os passos de 1 a 3 da Figura 4.4 várias vezes. Antes de executar o passo 4, os conjuntos de vértices marcados como derramados associados a cada função de heurística são avaliados; o conjunto efetivamente derramado é aquele que causa menor custo total.

O algoritmo também pode ser melhorado em outro aspecto. Durante o processo de simplificação há uma atitude muito pessimista em relação à colorabilidade do grafo. Em vários casos é possível fazer a coloração mesmo que o número de vizinhos seja maior que R_f. Reutilizando cores sempre que possível, pode-se gerar colorações onde vários vizinhos de um vértice v, originalmente com grau maior que R_f, recebam cores iguais, facilitando a coloração de v. Propõe-se aqui que o passo de coloração seja tentado mesmo quando derramamento foi considerado necessário. Se o processo de coloração consegue encontrar a cor para um vértice marcado como derramado, então nenhum derramamento é, de fato, gerado para aquele vértice.

O emprego deste melhoramento pode ser particularmente interessante em colorações envolvendo candidatos de tamanhos distintos, visto que a suposição de que todos os vizinhos receberão cores distintas é mais crítica nestas situações. Um exemplo da aplicabilidade do mecanismo pode ser observado no grafo da figura 4.3(a), onde o algoritmo é capaz de encontrar uma 3-coloração sem gerar derramamento.

Chaitin não apresenta análise do percentual de otimização obtido com a aplicação de seu algoritmo.

4.4.2 Alocação por Coloração baseada em Prioridades

Chaitin pressupõe que a máquina alvo da alocação é LOAD/STORE, sendo por isto, sempre benéfico manter variáveis em registradores. Em máquinas CISC é possível especificar operandos em

memória dentro das instruções, de modo que nem sempre há vantagens em utilizar registradores. A geração de derramamento antes e depois de cada referência também tende a diminuir a qualidade do código objeto quando existe uma grande pressão por registradores apenas em algumas áreas do procedimento.

Chow e Hennessy [CH84, CH90] desenvolveram um algoritmo que expande o algoritmo original de Chaitin em vários aspectos, notadamente os dois acima mencionados.

O Grafo de Interferências

A questão do derramamento envolve um compromisso com o tempo a ser gasto no processo de alocação. O desejável é dividir uma cadeia incolorável em duas (ou mais) outras coloráveis gerando a menor quantidade de derramamento. Infelizmente este também é um problema *NP-completo* no número de pontos presentes na cadeia a ser dividida [LH84]. A derivação de um novo conceito, trecho de vida, ajuda a reduzir o tempo gasto no processo de divisão. Um trecho de vida de uma variável v, $tv(v)$, é formado por um conjunto de blocos básicos que contêm pontos pertencentes à vida de uma variável.

Neste algoritmo um candidato a alocação é um par composto por uma variável e um de seus respectivos trechos de vida, tv. Inicialmente nenhum esforço é feito para transformar cada cadeia de uma variável em um candidato diferente; cada vértice representa um candidato com a totalidade da vida de uma variável. Variáveis derramadas são submetidas a um alocador que trabalha sobre blocos básicos, empregando alguns registradores específicos para este propósito.

Sendo um registrador associado a um trecho de vida, a noção de interferência também deve ser modificada. Dois candidatos interferem entre si, se eles têm um bloco básico em comum. Uma conseqüência deste conceito relaxado de interferência é que duas variáveis nunca vivas simultaneamente em qualquer ponto do procedimento podem ser consideradas conflitantes.

Tanto a convenção inicial sobre o tamanho do trechos de vida, quanto a noção de interferência tendem a dificultar a coloração do grafo. Por outro lado, estas convenções têm efeitos positivos sobre o tempo de execução do algoritmo de coloração. Primeiro porque o cálculo de interferências fica restrito à interseção de blocos básicos. Segundo, porque, de acordo com a Tabela 4.1 e o Apêndice A, as máquinas mais recentes, em geral, possuem registradores suficientes para colorir os grafos gerados pela maioria dos procedimento reais, sem que nenhum derramamento seja necessário. Desta forma o custo inicial de quebrar a vida da variável em suas várias cadeias é evitado; se for caso, o processamento de derramamento se encarrega desta tarefa.

Coloração

O processo de coloração é progressivo, ou seja, em cada iteração do algoritmo uma cor é associada a um candidato. Antes de iniciar o processo, todos os vértices com grau menor que R_f são removidos do grafo para coloração *a posteriori*. A remoção destes vértices é recomendável porque a coloração de um deles, que são denominados irrestritos, pode prejudicar a coloração daqueles que têm grau maior que R_f (restritos). A associação de uma cor a um vértice faz esta cor inutilizável em todos os seus vizinhos.

A escolha do candidato a colorir em cada iteração é baseada em prioridades. A prioridade ($LUCRO$) é uma estimativa da redução que a promoção do candidato a um registrador traz ao tempo de execução do procedimento. Os fatores influenciando o lucro são o número de referências

ao candidato feitas ao longo de seu trecho de vida (tv) e o próprio tamanho do trecho de vida. Candidatos com maiores trechos de vida devem ser penalizados, pois, em tese, tornam mais difícil a coloração do grafo (isto é mais ou menos equivalente à normalização pelo grau do vértice, na função heurística de Chaitin).

Algumas medidas de custo devem ser estabelecidas antes da definição da função $LUCRO$:

MC: custo de fazer um LOAD ou STORE;

LS: lucro de buscar um dado em registrador e não na memória;

SS: lucro de armazenar um dado em registrador e não na memória.[6]

Seja $u(c,b)$ e $d(c,b)$ o número de usos e definições, respectivamente, do candidato c dentro do bloco básico b. O número de blocos básicos de um candidato c é dado por $Nb(c)$. Finalmente uma última observação: a divisão de cadeias, detalhada adiante, pode fazer com que um LOAD e/ou um STORE sejam colocados no início e/ou fim de alguns blocos básicos pertencentes ao trecho de vida do candidato. $ls(c,b)$ expressa o número de LOAD e/ou STORE do candidato c dentro de b. $ls(c,b)$ varia de zero a dois.

A função de lucro pode então ser definida:

$$LUCRO(c) = \frac{\sum_{\forall b \in tv(c)} ((LS * u(c,b) + SS * d(b,c) - MC * ls(c,b)) * F(b))}{Nb(c)}$$

O que a função calcula é o lucro com cada referência a c, ponderado pela freqüência estimada de execução do bloco onde ela acontece. O custo do código de derramamento (ls) e a normalização pelo tamanho do trecho de vida do candidato completam a heurística.

A associação de uma cor a um candidato pode deixar vários de seus vizinhos incoloráveis ou seja, todas as cores são inutilizáveis em alguns vértices. Os candidatos incoloráveis são divididos em trechos de vida menores, cada um deles colorável. O processo total se repete. A Figura 4.5 apresenta o algoritmo.

1. Selecione o conjunto de candidatos irrestritos;
2. Repita até que todos os candidatos restritos recebam cor:
   (a) Para todo candidato c, Calcule $LUCRO(c)$ se ainda não calculado;
   (b) Remova candidatos c, tais que $LUCRO(c)$ é negativo;
   (c) Escolha um candidato (entre os restritos) com maior $LUCRO$ e dê-lhe uma cor ainda não utilizada nos seus vizinhos;
   (d) Se algum candidato tornou-se incolorável, divida seu trecho de vida, de acordo com as regras de divisão;
   (e) atualize o conjunto de irrestritos;
3. Associe cores aos vértices irrestritos.

Figura 4.5: Alocação por coloração baseada em prioridades.

[6] Em máquinas LOAD/STORE as três grandezas são aproximadamente iguais.

O conjunto de irrestritos é afetado pelo processo de divisão. Supondo que dois candidatos são gerados na divisão, ambos podem interferir com algum dos irrestritos tornando-o restritos. Conversamente algum (ou ambos) dos novos candidatos pode ser irrestrito. Justifica-se, então, o passo 2(e) na Figura 4.5.

Gerenciamento de Derramamento

Durante a divisão de um candidato c, o maior componente conexo do trecho de vida que permanece colorável é retirado de c. Seja $tv(c')$ o trecho de vida retirado de $tv(c)$. Se o trecho de vida resultante da remoção de $tv(c')$ de $tv(c)$ continua incolorável, o processo é repetido sobre ele. A estratégia de divisão evita a proliferação de pequenas cadeias, o que resultaria em muito código de derramamento nas extremidades dos trechos de vida recém-formados. A Figura 4.6 apresenta em detalhes o algoritmo de divisão.

1. Encontre um bloco básico colorável em $tv(c)$, preferivelmente um que inicia uma cadeia, e o atribua a $tv(c')$; se nenhum pode ser encontrado, remova o candidato c do grafo de interferências; o gerador de código se encarrega do derramamento nestes casos;
2. Adicione blocos básicos a $tv(c')$ enquanto ele é mantido conexo e colorável;
3. Faça $tv(c)$ igual aos blocos básicos (do trecho de vida original) não pertencentes a $tv(c')$;
4. atualize as interferências de $tv(c)$ e $tv(c')$ no grafo;
5. Se $tv(c)$ ainda é incolorável repita todo o processo.

Figura 4.6: Divisão de trechos de vida de uma variável.

A noção de derramamento é resultante das seguintes observações:

- sempre que a primeira referência a um candidato c dentro de um bloco básico b for um uso e existir um predecessor de b fora do trecho de vida de c, então um LOAD de c deve ser emitido na entrada de b;
- se um LOAD do candidato c está presente na entrada de um bloco básico b, então todos os predecessores de b pertencentes ao trecho de vida de c devem emitir um STORE de c nas suas saídas;
- Se o candidato c está vivo na saída de um bloco b que tem pelo menos um sucessor não pertencente ao trecho de vida de c, então um STORE deve ser inserido na saída de b.

A partir das informações de LOAD e/ou STORE acima é possível calcular para cada bloco básico b, no trecho de vida de vida do candidato c, o valor de $ls(c, b)$ utilizado na função $CUSTO$. Uma outra fonte de derramamento acontece quando é impossível realizar o passo 1 da Figura 4.6, ou seja, todos os blocos básicos do trecho de vida do candidato são incoloráveis. O candidato é, nestes casos, removido do grafo de interferências e submetido a alocação local (dentro de cada bloco básico).

No passo 2 da Figura 4.6, Larus e Hilfinger [LH84] recomendam que a adição de blocos básicos ao trecho de vida sendo formado deve seguir um critério de busca em largura, pois tende a minimizar

o efeito do derramamento. A aparente justificativa para este fato é que variáveis vivas em um ramo de um condicional e no bloco básico que o sucede (o condicional) também estarão vivas no outro ramo. Dado que a inserção de um dos ramos e do sucessor implica em inserir o outro ramo ou gerar código de derramamento no ramo já inserido, é mais conveniente tentar antes a inserção de ambos os ramos.

Comentários

Um esquema mais refinado poderia ser usado para selecionar a cor a atribuir a um vértice, considerando o efeito da escolha em relação à colorabilidade dos vizinhos ainda não coloridos. Por exemplo, se existem duas cores (azul, verde) disponíveis para coloração de um vértice v e a cor azul já é inutilizável na maioria dos seus vizinhos, então é preferível utilizar azul em v, dando maior oportunidade para as associações posteriores.

Chow reconheceu que em programas com grandes blocos básicos é possível que o número de interferências seja muito grande, o que pode aumentar consideravelmente o número cromático. Por isto, sempre que o número de instruções por bloco básico atinge certo número, o alocador força uma divisão de bloco básico.

Existe um conjunto de refinamentos que aumentam a qualidade do código. Alguns são abordados na seção 4.5.

Quando alocando registradores a candidatos de tamanhos diferentes, as mesmas observações feitas para a simplificação do grafo no método de Chaitin valem aqui na determinação dos candidatos irrestritos. A função *LUCRO* também pode ser alterada de modo similar.

Um dos problemas inerentes a arquiteturas RISC é reorganização de código. Embora o cálculo de interferências tendo por base um bloco básico tenha o efeito de aumentar o número cromático do grafo, ele permite uma maior liberdade na movimentação das instruções porque a contenção nos recursos, neste caso registradores, diminui. Esta questão será abordada no capítulo 6.

Chow fez várias análises sobre o percentual de otimização obtido com seu algoritmo. A avaliação completa pode ser encontrada em [CH90, LH84]. Seus experimentos foram feitos com 21 registradores disponíveis ao alocador. Em resumo, os números revelaram que a alocação reduz o tempo de execução em até 40%, com média (geométrica) entre vários *benchmarks*, inclusive compiladores, de 28% (sobre a total ausência de alocação). Um outro resultado interessante é que o percentual de otimização cai muito lentamente quando o número de registradores alocáveis é reduzido. Tal comportamento se verifica enquanto existem pelo menos oito registradores disponíveis.

4.4.3 Alocação por Coloração Hierárquica

A técnica de Chow e Hennessy melhora a geração de código de derramamento, mas não considera a estrutura do fluxo de controle do procedimento. É possível, por exemplo, que blocos básicos dentro de malhas sejam pontos de divisão do trecho de vida de um candidato, mesmo que o candidato nunca seja referenciado naquela malha. Nestas situações é provável que o custo de inserção de derramamento no local errado penalize candidatos que, potencialmente, trariam vantagens se alocados a registradores.

Callahan e Koblenz [CK91] desenvolveram um alocador sensível à estrutura do procedimento. A idéia principal é dividir o grafo de controle de fluxo do procedimento em um conjunto de coleções de blocos básicos, denominados *tiles*, que cobre o grafo. *Tiles* normalmente representam ou uma

malha ou uma estrutura condicional e são organizados hierarquicamente, com um *tile* contendo outros *tiles*. A identificação de *tiles* é baseada em análise de intervalos [ASU86], mas este processo não será detalhado. Como exemplo, considere o trecho de programa mostrado na Figura 4.7, onde existem três *tiles*: T0, T1, T2. *Tiles* ou são disjuntos, ou um é parte estritamente contida em outro. Dados dois *tiles*, s e t, tal que $s \subset^7 t$ e não existe *tile* t' que satisfaz a $s \subset t' \subset t$, então s é um *subtile* de t e, t o *supertile* de s. O conjunto dos *subtiles* de t é dado por $subtiles(t)$. Os blocos básicos de t que não pertencem a qualquer dos *subtiles* são denotados por $bt(t)$. Na Figura, o conjunto $bt(T0)$ é formado pelos blocos básicos rotulados com `ret`, `b0`, `b1`. T1 e T2 são *subtiles* de T0. Em princípio, a unidade de alocação é um *tile*. Os blocos fictícios denotados por `b0` e `b1` são criados pelo processo que identifica *tiles*. Eles são necessários porque se código de derramamento for gerado para um *tile*, devem existir pontos de entrada e saída do *tile*, onde tal código pode ser colocado.

Tiles são visitados de maneira ascendente. Para todo *tile*, um grafo de interferências local é construído e colorido usando pseudo-registradores em número igual ao de registradores físicos disponível. Com isto as decisões de derramamento são feitas localmente. O resultado da alocação vai sendo propagado até os níveis mais altos da árvore de *tiles*, até que se encontre a raiz, que representa o procedimento inteiro.

Em um segundo estágio os registradores físicos serão associados aos pseudo-registradores de maneira descendente. Código de derramamento é inserido durante este estágio. A manutenção de associações prévias entre *tiles* e *subtiles* também é respeitada, sempre que possível.

O Grafo de Interferências

Cada candidato à alocação é uma cadeia de uma variável. Um candidato é local a um *tile* t se todas as referências a ele acontecem dentro dos blocos básicos pertencentes a $bt(t)$; se o candidato é referenciado em t, mas não é local, ele é global a t. Para cada *tile*, um grafo de interferências é construído. A coloração de um destes grafos propaga as seguintes informações ao *supertile*:

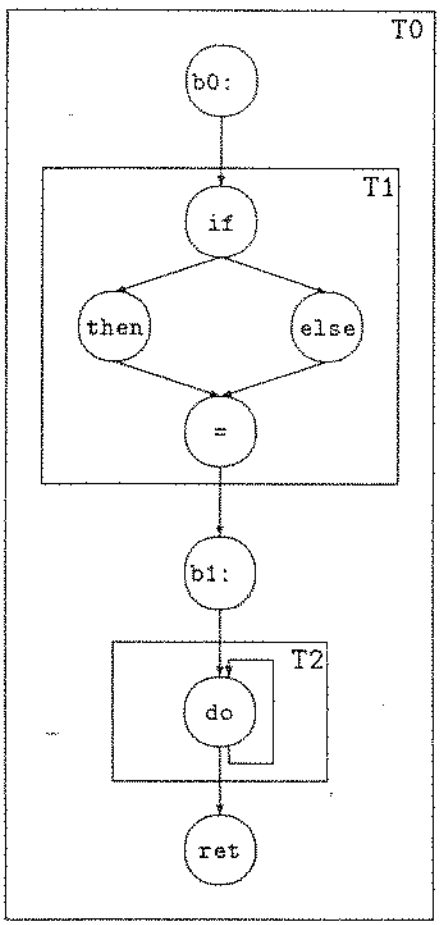
- Os candidatos locais ao *tile* que receberam pseudo-registradores; mais precisamente, para cada pseudo-registrador, um novo candidato que combina todos os candidatos locais que foram associados a aquele pseudo-registrador. Estes novos candidatos são denominados candidatos de sumário do *tile*;
- Para todo candidato global que recebeu um pseudo-registrador, todos os seus conflitos (dentro do *tile*) com os candidatos globais e de sumário daquele *tile*.

Os candidatos a alocação dentro de um *tile* t são de dois tipos:

- variáveis referenciadas dentro de $bt(t)$;
- candidatos de sumário de cada um dos *subtiles* de t;

Variáveis vivas, mas não referenciadas em t, não participam do grafo de interferências deste *tile*. A presença de candidatos de sumário dos *subtiles* no grafo de interferências do *tile* t é para garantir que os candidatos promovidos a pseudo-registradores em t podem cruzar os *subtiles* sem gerar código de derramamento.

[7] leia-se: é parte de.



(a)

```
int p (x, y)
int x, y;
{
     .
     .
     .
  if (x > y)
    ret = x;
  else
    ret = y;
  ret = 2 * ret;
  do
    ret = ret + y;
  while (--x);
  return(ret);
}
```

(b)

Figura 4.7: Divisão do grafo de controle de fluxo de um programa em *tiles*.

Candidatos de sumário de *subtiles* podem eventualmente ser considerados derramados quando participam do grafo de interferências do *tile*. O derramamento de um destes candidatos indica que há uma forte pressão por registradores na região do *tile* também pertencente ao *subtile* e que candidatos de sumário tem menor vantagem de serem promovidos a registrador. Como as decisões de derramamento no *subtile* já foram feitas, cabe ao segundo estágio da associação de registradores corrigir este problema, possivelmente gerando mais derramamentos no *subtile*.

As arestas resultam de quatro situações:

1. conflitos existentes no conjunto *bt* para o *tile*. A falta de informação global, requer que dois candidatos vivos em um mesmo ponto sejam considerados conflitantes;
2. conflitos propagados por candidatos globais que receberam pseudo-registrador nos *subtiles* e também pertencem ao grafo do *tile*;
3. conflitos entre candidatos de sumário de cada *subtile*;
4. conflitos entre todos os candidatos de um *subtile* que receberam pseudo-registradores (sumário e globais) e candidatos referenciados no *tile* que são vivos, mas não referenciados no *subtile*.

Coloração

A heurística de coloração do grafo de interferências segue o mesmo critério empregado por Chaitin [Cha82a]: simplificação e coloração. A coloração é repetida até que uma R_f-coloração seja obtida. No primeiro estágio usando pseudo-registradores, no segundo registradores físicos.

A associação final de registradores exige a reconstrução do grafo de interferências para cada *tile*, partindo do *tile* raiz. Durante a primeira fase, candidatos não referenciados no *tile* t não participavam do grafo de interferências de t. Associações corretas necessitam incluir estes candidatos, desde que eles estejam vivos no *tile* e tenham recebido registrador no *supertile*. De outro modo, os seus conteúdos poderiam não ser mantidos. Se alguns dos candidatos no *tile* receberam registradores no *supertile*, é recomendável manter as associações. O mecanismo de preferência, apresentado na próxima seção, é utilizado para priorizar tais associações.

Gerenciamento de Derramamento

Em oposição ao método de Chaitin, existem várias situações onde alguma espécie de derramamento é necessária:

- um *tile* mantém o candidato em registrador e um *subtile* o mantém na memória; no bloco básico de entrada (saída) do *subtile* o candidato deve ser armazenado (restaurado), e vice-versa;
- ambos, *tile* e *supertile*, associam registrador ao candidato, mas os registradores são distintos. Nos blocos básicos de entrada e saída do *tile*, operações de cópia entre os registradores são necessárias;
- uma variável que não recebe registrador dentro de um *tile* é carregada antes de cada uso e armazenada após cada definição.

A escolha do candidato a ser derramado, durante um bloqueio do processo de simplificação do grafo de interferências relativo a um *tile* t, é baseada no número de referências ao candidato dentro de t e na interação entre t, seus *subtiles* e o *supertile*. A variável com menor lucro de alocação é derramado dentro do *tile*.

O lucro de um candidato c, devido às referências a c dentro de t, é definido de modo similar à função $LUCRO$ do método de Chow:

$$LUCRO_r(c,t) = \sum_{\forall b \in bl(t)} (LS * u(c,b) + SS * d(b,c)) * F(b)$$

A associação de c a um registrador em t é estimulada se c também está em registrador dentro de algum *subtile* s de t. $Reg(c,s)$ denota que o candidato c foi alocado a um registrador em s, situação onde vale 1 (0, se não). Conversamente, há uma penalidade quando c está em memória em algum *subtile* s, $Mem(c,s)$.

A penalidade (ou estímulo) é basicamente o custo de armazenar e restaurar c nos blocos básicos de entrada e saída de cada *subtile* s, $bb_e(s)$ e $bb_s(s)$, respectivamente. Seja $TRANSF(c,s)$ este custo:

$$TRANSF(c,t) = MC * (F(bb_e(s)) + F(bb_s(s)))$$

Portanto, o lucro total de alocação pode ser medido por:

$$LUCRO_t(c,t) = LUCRO_r(c,t) + \sum_{\forall s \in subtiles(t)} (Reg(c,s) - Mem(c,s)) * TRANSF(c,s)$$

Observe-se, entretanto, que o valor de $LUCRO_r(c,t)$ é nulo quando c é um candidato de sumário de um *subtile* s, pois não há referências a c em $bl(t)$. Para estes candidatos, $LUCRO_t(c,t)$ é definido como $LUCRO_t(c,s)$.[8]

Na primeira fase da associação, um candidato global também é automaticamente derramado no *tile* t se $(LUCRO_t(c,t) + TRANSF(c,t) < 0)$. Nestes casos, a "desvantagem" de manter c em registrador dentro de t (possivelmente causada porque o candidato está na memória em vários *subtiles*) supera o custo de transferência, mesmo se o *supertile* decidir promover c a registrador.

O processo de associação final ainda pode fazer mais derramamento: se c está em registrador no *tile* t, na memória no *supertile* de t e $(LUCRO_t(c,t) < TRANSF(c,t))$, então c é derramado em t, pois sua alocação não justifica o custo de transferência memória/registrador.

Comentários

O gerenciamento de derramamento é feito muito eficiente, mas os autores não fazem considerações a respeito dos efeitos práticos da implementação, nem sequer mencionam se ela existe. De qualquer modo, espera-se que os resultados sejam melhores em programas que fazem muita computação de ponto flutuante, visto que nestes os blocos básicos maiores tendem a gerar mais temporários e aumentar a pressão por registradores em apenas algumas áreas do programa.

De certa forma este algoritmo integra os métodos de Chaitin e de Chow, quando determina o candidato a ser derramado através da (minimização da) função LUCRO. Se anexado a isto, a

[8] Precisamente, o valor é o somatório da função $LUCRO_t(v,s)$ para cada um dos candidatos v que formam o candidato de sumário.

tentativa a coloração for feita mesmo em situações onde derramamento foi considerado necessário (vide seção 4.4.1), os métodos de Chow e Chaitin ficam praticamente iguais (a menos do tratamento de derramamento).

Um outro aspecto positivo é que o algoritmo pode ser executado eficientemente em máquinas com mais de um processador, considerando que a alocação em *tiles* disjuntos pode ser executada em paralelo.

4.5 Particularidades no Uso de Registradores

Por razões de eficiência, um conjunto de registradores é reservado à passagem de parâmetros entre procedimentos e outro para retorno de valores. Registradores de uso geral, mas com particularidades no uso, tais como base de endereçamento e contador de malhas, são comuns em arquiteturas mais complexas. Em qualquer dos casos o alocador deve respeitar convenções e/ou restrições. Uma outra questão associada a chamada de procedimentos é gerenciamento de contexto: se variáveis residentes em registradores estão vivas durante uma chamada de um procedimento é necessário armazenar e posteriormente restaurar o valor do registrador da memória. Visando diminuir o custo de cruzar chamadas, os registradores são tradicionalmente divididos em dois subconjuntos, ou categorias, fixos:

PRESERVADOS:[9] são registradores que devem ser armazenados no prólogo e restaurados no epílogo de cada procedimento, se utilizados. Dentro do procedimento o uso é livre, podendo inclusive cruzar chamadas sem incorrer em qualquer custo no chamador;

VOLÁTEIS:[10] são registradores que para serem usados em um procedimento, devem respeitar a condição de nunca estarem vivos durante chamadas. Para garantir esta condição o compilador deve, eventualmente, inserir código para manter os valores dos registradores cruzando chamadas.

Estabelecida uma divisão, o alocador deveria atribuir prioridades às categorias, de modo que cada variável é tentativamente associada a um registrador da categoria mais apropriada. Ambas as categorias têm vantagens. Para variáveis que cruzam muitas chamadas, um registrador preservado é a melhor opção, pois ele só será armazenado se o chamado o utilizar. Temporários gerados pelo compilador, normalmente empregados na avaliação de expressões, são inerentemente identificados com registradores voláteis.

Nesta seção examina-se como os alocadores apresentados na seção anterior podem ser adaptados para tirar vantagem das oportunidades de otimização que estas convenções oferecem.

Preferências entre Registradores

Chaitin usou *subsumption* para evitar cópias entre variáveis. A desvantagem de adotar este método é que, em certos grafos de interferências, derramamento pode passar a ser necessário em função do seu uso. Um conceito similar e mais flexível é *pré-coloração*. Pré-colorir um candidato equivale a avisar o alocador que aquela associação é recomendada. Existem dois tipos de situação onde pré-coloração é útil:

[9] do inglês: *callee-saved*
[10] do inglês: *caller-saved*

- Se existe uma cópia entre um registrador físico e um candidato, então o candidato passa a ser pré-colorido com aquele registrador físico;
- Se existe uma cópia entre candidatos, cada um deles é pré-colorido com o outro;

O primeiro tipo soa incoerente com a suposição de que todos os candidatos estão inicialmente associados a registradores simbólicos. Todavia, para tratar certas convenções de uso de registradores ou assimetrias de uma arquitetura pode ser conveniente gerar código que opera sobre registradores físicos; a próxima subseção dá um exemplo. A presença de registradores físicos no código intermediário requer uma extensão no grafo de interferências: além dos candidatos originais, todos os registradores físicos passam também a ser vértices, mas presença deles é devida exclusivamente à representação de conflitos com os candidatos reais. Vértices representando registradores físicos nunca são removidos do grafo de interferências e sua coloração é, obviamente, pré-definida.

Um candidato pode ser pré-colorido com vários outros candidatos ou registradores físicos, formando uma lista de pré-coloração. Durante o processo de associação de registradores, um candidato que possui um registrador físico na sua lista de pré-coloração recebe preferencialmente o registrador correspondente à pré-coloração. Quando um dos vizinhos do candidato já foi associado ao registrador preferido, uma outra pré-coloração é tentada, se existir. A associação de registrador a um candidato não pré-colorido tenta respeitar, se possível, as pré-colorações dos vizinhos ainda não associados.

A associação de um registrador a um candidato c, que está na lista de pré-coloração de outros candidatos tem o efeito de substituir as ocorrências de c, em todas as listas de pré-coloração, pelo registrador físico que lhe foi associado.

Passagem de Parâmetros em Registradores

Em um algoritmo que calcula interferências a nível de pontos de execução, como o de Chaitin, a interação entre o gerador de código e o alocador torna fácil a geração de código eficiente para passagem de parâmetros. Se determinado parâmetro p_i, é recebido em um registrador r_i, este registrador é a melhor opção, em termos de custo, para acesso ao parâmetro ao longo do procedimento. Na presença de novas chamadas é possível que o parâmetro efetivo não possa ser mantido no mesmo registrador. Mais ainda, é possível que o *i-ésimo* parâmetro de entrada seja o *j-ésimo* parâmetro de uma chamada a outro procedimento. Em qualquer das situações é necessário emitir código correto e ao mesmo tempo priorizar as associações que aumentam a qualidade do código.

Uma solução simples e eficaz para a questão abordada acima é emitir, no código intermediário, uma instrução de cópia dos registradores físicos que inicialmente contêm os parâmetros para registradores simbólicos. A partir daí todas as referências a parâmetros, dentro do procedimento, seriam feitas via os registradores simbólicos. Quando o alocador encontra a cópia entre os registradores, faz uma pré-coloração dos simbólicos com os físicos. Usando o esquema apresentando na subseção precedente, a pré-coloração pode ter sucesso e, na fase de emissão do código objeto, um simples otimizador *peephole* remove a operação de cópia. O mesmo se aplica a parâmetros de saída e valores retornados por funções.

Considere, por exemplo, o procedimento (função) escrito em linguagem C na parte (a) da Figura 4.8. A parte (b) mostra uma possível representação intermediária. Os registradores físicos (`r1`, `r2`, `r3`) são usados para passagem de parâmetros e `r_v` é o registrador simbólico representando a variável v. O grafo de interferências relativo ao código intermediário pode ser observado na

```
p(x, y, z)
int x, y, z;
{
   q(x, 2);
   r(z, y);
}
```

(a)

```
bb1:  mov  r1, r_x
      mov  r2, r_y
      mov  r3, r_z
bb2:  mov  r_x, r1
      mov  2, r2
      call q
bb3:  mov  r_z, r1
      mov  r_y, r2
      call r
```

(b)

Figura 4.8: (a) Um programa em C e sua possível representação intermediária (b).

Figura 4.9(a). A coloração do grafo, respeitando as pré-colorações (indicadas na Figura ao lado de cada candidato), gera as seguintes associações: {(r_x, r1),(r_y, r4), (r_z, r3)}. O código final é mostrado na Figura 4.9(b).

Este esquema não pode ser usado diretamente no método de Chow porque as interferências dentro dos blocos básicos impediriam qualquer sucesso no processo de pré-coloração (como até agora definido). A solução é usar uma estratégia mais restrita. A geração de código intermediário e a geração de pré-colorações permanecem iguais, mas registradores físicos não participam do grafo de interferências.

Durante o processo de associação, as pré-colorações de um candidato são estritamente respeitadas pelos demais, pois cada pré-coloração passa a indicar que o o registrador é usado em alguma parte do trecho de vida do candidato.[11] Em outras palavras, é como se cada candidato recebesse vários registradores: um decorrente da associação e outros das pré-colorações. Conflitos entre registradores de ambos os conjuntos de pré-coloração de dois vizinhos não resultam em inconsistências porque um único registrador nunca é associado a dois conjuntos de pré-coloração em um único bloco básico (lembre que chamadas de procedimentos delimitam blocos básicos).

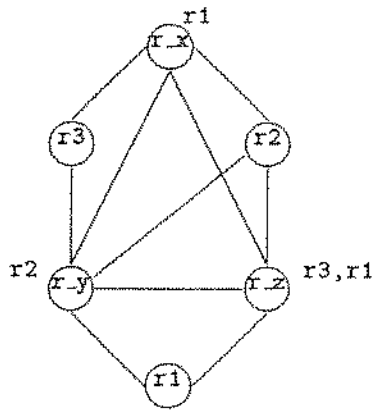
Preservados *vs* Voláteis

A questão relevante na escolha da categoria do registrador a associar a um candidato é o número de chamadas que ele cruza. O método de Chaitin pode usar diretamente o conceito de pré-coloração para priorizar associações de um candidato a uma ou outra classe de registradores. Se uma ou mais chamadas são cruzadas por um candidato, todos os registradores preservados são associados ao conjunto de pré-coloração do candidato. Caso contrário, o conjunto de voláteis é que seria associado.

No método de Chow é possível ser mais específico e refinar a função $LUCRO$ de um candidato c, em três outras :

- $LUCRO_v$: lucro de associar o candidato a um registrador volátil. Para todo bloco básico b, onde o candidato está vivo durante uma chamada, o valor de $ls(c, b)$ é aumentado de acordo,

[11] A rigor, para cada vizinho, o candidato só necessita respeitar as pré-colorações dos blocos básicos que geram a interferência.



(a)

```
bb1:  mov r2, r4
bb2:  mov 2, r2
      call q
bb3:  mov r3, r1
      mov r4, r2
      call r
```

(b)

Figura 4.9: (a) O grafo de interferências referente ao código intermediário da Figura 4.8 e (b) o código final após a associação de registradores.

refletindo a necessidade de armazenar (restaurar) o registrador antes (depois) da chamada.

- $LUCRO_p$: lucro de associar o candidato a um registrador preservado. O valor da função $LUCRO$ original deve ser reduzido para caracterizar a necessidade de armazenar e restaurar o conteúdo prévio do registrador.
- $LUCRO_l$: É a própria função $LUCRO$ definida anteriormente. Se existe um registrador preservado já utilizado nas associações anteriores, mas disponível ao candidato, não é necessário computar o custo de gerenciá-lo na entrada e saída do procedimento.

Para determinar o candidato de maior lucro para alocação, as três funções são consideradas, a maior escolhida. Pode acontecer que um registrador na categoria de maior lucro não esteja disponível para a associação. Se este é o caso, o processo de escolha é repetido sem considerar aquela função. Em caso de igualdade entre os valores, prioridade deve ser dada a $LUCRO_p$, pois cria oportunidades de $LUCRO_l$ vir a ser utilizada posteriormente.

4.6 Conclusões

O uso de registradores reduz o tráfego de dados entre memória e processador e indiretamente também diminui o tráfego devido a instruções, pois podem ser especificados em menor espaço. Desde que estas reduções são fatores significativos no tempo de execução de programas, máquinas RISCs contêm um grande conjunto de registradores e um restrito esquema de busca de operandos em memória, o que exige técnicas robustas de associação de variáveis a registradores. Contudo, o problema de manter variáveis em registradores é antigo e sempre foi considerado complexo.

Algumas linguagens, tais como C e Bliss, permitem que o programador ajude o compilador na alocação especificando algumas variáveis para residirem em registradores. Este método, todavia, é

dependente da habilidade do programador reconhecer as variáveis mais utilizadas e, eventualmente, requer alguma espécie de *profile* para alcançar resultados satisfatórios.

Mais recentemente, otimizadores de código têm dado suporte ao emprego de melhores técnicas de alocação, notadamente alocação por coloração de grafos. Neste capítulo, foi mostrado como este modelo pode ser usado efetivamente, bem como sua adequação a uma variedade de convenções que aumentam a qualidade do código objeto final. Embora as convenções sejam fontes valiosas de otimização, a literatura tem pouca documentação sobre seu emprego em conjunto com alocação por coloração.

Capítulo 5

Alocação Interprocedimental de Registradores

5.1 Introdução

Alocação intraprocedimental de registradores tem embutido um custo de reutilização dos registradores resultante do gerenciamento do contexto existente durante uma nova chamada. Alocação interprocedimental é a tentativa de eliminar este custo. Garantir que procedimentos diferentes utilizem registradores distintos é, em princípio, uma condição suficiente para eliminar qualquer gerenciamento de registradores devido a chamadas de procedimentos.

Considere, por exemplo, o grafo da Figura 5.1 onde vértices representam procedimentos e arestas chamadas. Sejam $(2,3,4)$ os números de registradores utilizados nos procedimentos (A,B,C), respectivamente. Se existem nove, ou mais, registradores físicos disponíveis $(r1,r2,...,r9)$, a tarefa de um alocador interprocedimental é associar registradores seguindo um critério que leve a um resultado final como $((r1,r2),(r3,r4,r5),(r6,r7,r8,r9))$.

Mais precisamente, um alocador interprocedimental deve tratar dois problemas decorrentes da alocação intraprocedimental:

- Variáveis locais em procedimentos distintos podem ser associadas ao mesmo registrador. A cada nova chamada o compilador insere código para permitir reutilização de registradores;
- Variáveis globais ao programa podem ser associadas a registradores diferentes em vários procedimentos. O compilador usa a memória como interface para comunicação entre dois procedimentos usando a mesma variável.

O tratamento destes problemas exige a movimentação de dados entre registradores e memória a cada nova chamada. Nas arquiteturas RISCs, em particular, o acréscimo no tempo de execução associado a esta tarefa pode ser significativo quando programas são intensivos em chamadas. Desta forma, utilizar um grande número de registradores pode até mesmo contribuir negativamente no contexto de otimizações, quando critérios de gerenciamento não são bem estabelecidos.

A divisão de registradores nas categorias voláteis e preservados resolve parcialmente o problema, mas a ausência de informação interprocedimental pode continuar a gerar tráfego para (de) a memória desnecessariamente. Ao armazenar um registrador volátil, o procedimento chamador o

Figura 5.1: Chamadas de procedimento em um programa simples.

faz na suposição de que pelo menos um dos seus chamados usá-lo-á, mas este pode não ser o caso. Similarmente, quando um procedimento usa um registrador preservado deve manter o seu conteúdo inicial, pois um chamador pode estar esperando este comportamento. De novo, o caso contrário não é impossível. Mas, na falta de informação, o pior caso é sempre suposto.

Por outro lado, a aplicação de algoritmos de alocação intraprocedimental baseados em coloração de grafos tem mostrado que, em média, poucos registradores são necessários para manter todas as variáveis simples de um procedimento fora da memória. Conclui-se, portanto, que um aumento no número de registradores de uma arquitetura, normalmente, não diminui proporcionalmente o tempo de execução de programas se alocados independentemente a cada procedimento.

Alocação interprocedimental é, então, motivada pela necessidade de fazer bom uso dos grandes conjuntos de registradores que possivelmente equiparão as máquinas do futuro. No presente, o custo de gerenciar o contexto de registradores entre chamadas de procedimentos e o baixo número de registradores utilizados por procedimento têm servido para desencadear a pesquisa na área.

O principal obstáculo à aplicação da alocação interprocedimental é a sua incompatibilidade com ambientes de desenvolvimento com suporte a módulos compiláveis separadamente. Indiscutivelmente, este mecanismo de abstração aumenta a produtividade durante o desenvolvimento e é oferecido pela maioria das linguagens de programação mais recentes. O problema é que os resultados da alocação interprocedimental são globalmente afetados por mudanças locais. A alteração de um procedimento muda os seus requerimentos com respeito a uso de registradores, isso por sua vez muda os registradores disponíveis para alocação nos demais. Uma simples modificação pode, portanto, ter o efeito de desencadear uma recompilação (realocação no melhor dos casos) de outros procedimentos e/ou módulos.

Este capítulo trata de técnicas para minimizar o custo associado às situações acima, especialmente a primeira. As seções seguintes abordam soluções por *hardware* e *software*. Uma outra é dedicada a compará-las. As conclusões sobre o assunto encerram o capítulo.

Terminologia

Um *grafo de chamadas estático* expressa os relacionamentos entre procedimentos de um programa. Os vértices do grafo representam procedimentos; uma aresta orientada de v_i a v_j representa uma possível chamada, em tempo de execução, de v_j dentro de v_i. Ciclos no grafo representam eventuais *recursões* (diretas ou indiretas) no programa. Uma *chamada indireta* é aquela onde o procedimento alvo não é conhecido em tempo de compilação (por exemplo, através de um apontador ou parâmetro).

O *grafo de procedimentos* é formado pelo fecho transitivo do grafo de chamadas estático, e dois procedimentos podem estar *ativos simultaneamente* se existe uma aresta entre eles no grafo de procedimentos.

O *contexto dos registradores* em um ponto, ou simplesmente *contexto*, é o conjunto de registradores vivos naquele ponto.

5.2 Esquemas de Alocação em *Hardware*

O mecanismo de janelas deslizantes foi uma das primeiras tentativas de alocação interprocedimental. Sendo um novo conjunto de registradores automaticamente alocado a cada chamada, não há necessidade de gerenciar explicitamente o contexto de registradores durante chamadas. Mesmo um requisito mais forte é alcançado. Dois procedimentos podem, eventualmente, utilizar a mesma janela (os mesmos registradores) sem incorrer em qualquer custo de gerenciamento, mesmo estando relacionados no grafo de chamadas estático, para isto é bastante existirem situações onde os dois não estão presentes simultaneamente na cadeia dinâmica de chamadas.

Modelos de arquitetura memória-memória [Tan90] com mapeamento de partes da memória em *caches* ultra-rápidos também têm sido propostos para evitar alocação interprocedimental em tempo de compilação. Neste esquema a "alocação" consiste basicamente em mudar a região de memória mapeada no *cache* a cada nova chamada. Em geral, o cálculo do endereço físico e a busca nos diretórios do *cache* são processos seriais e por isso pouco atrativos em relação a registradores; as constantes atualizações do *cache* também representam, de certa forma, o custo associado ao gerenciamento dos registradores.

O suporte exclusivo à linguagem C permitiu eliminar muitos dos problemas de mapeamento em *cache* no projeto da máquina CRISP [Dit82]. Uma pilha de registradores contém os registros de ativação dos últimos procedimentos chamados. Sempre que uma instrução é buscada da memória para o *cache* de instruções, os endereços dos operandos relativos a variáveis locais escalares são convertidos para índices na pilha de registradores; a partir daí o índice funciona como um número de um registrador. Tendo pilha de 1024 registradores, as estatísticas mostram que são raras as situações onde é necessário mover registradores da pilha para a memória. Note-se que nenhuma alocação é necessária, todavia existe um problema de *relocação* quando um procedimento é chamado recursivamente.

Uma outra técnica proposta para implementação em *hardware* [SH89] exige que o chamador crie uma máscara com os registradores vivos no instante de cada chamada. Quando a chamada é feita, os registradores na máscara são automaticamente armazenados na memória. No retorno o inverso acontece. O problema aqui é que a instrução de chamada pode se tornar demasiadamente cara em relação às demais.

5.3 Esquemas de Alocação em *Software*

Intuitivamente, a primeira solução para alocação interprocedimental em *software* é construir um grafo de interferência global para o programa. Todas as variáveis do programa candidatas a receber registradores são consideradas vértices do grafo de interferência. Construído este grafo, um algoritmo de coloração dentre os apresentados nos capítulo precedente pode ser aplicado, gerando uma alocação interprocedimental.

A construção do grafo de interferência é dividida em três estágios: construção de grafos de interferências locais a cada procedimento, construção do grafo de procedimentos e geração do grafo global. Os primeiros resultam diretamente da análise de fluxo de dados dentro dos procedimentos.

Os vértices do grafo global são todos aqueles presentes nos grafos locais. As arestas vêm de interferências locais ou da seguinte situação: uma variável v de P está viva durante uma chamada a Q e existe uma aresta de P a Q no grafo de procedimentos. Neste caso v interfere com todas as variáveis de Q.

Em grafos de chamadas cíclicos (representando programas com recursão) existe a necessidade de tratar várias instâncias de uma mesma variável. Aplicar algoritmos intraprocedimentais diretamente não resolve o problema, já que a representação de uma recursão seria uma variável interferindo com ela mesma. Chamadas indiretas também exigem tratamento diferenciado, pois as arestas não são explicitamente representadas. Os algoritmos descritos adiante nesse capítulo fazem um preprocessamento no grafo de chamadas estático para tratar recursões e possíveis chamadas indiretas.

Mesmo em grafos acíclicos e sem chamadas indiretas, fazer alocação interprocedimental via coloração de um grafo de interferência global é impraticável. [SH89] aplicou a técnica a um programa com 220 procedimentos e 3500 linhas de código fonte escrito em Lisp. O grafo resultante teve por volta de 500 vértices e mais de 50.000 arestas. Aplicar um algoritmo intraprocedimental parece computacionalmente inaceitável, principalmente devido ao conhecimento prévio de que o número de registradores físicos disponível será insuficiente para manter todas as variáveis fora da memória. Some-se a isso que os dois principais algoritmos para alocação de registradores a partir de um grafo de interferência [Cha82a, CH90] tendem a ser lentos quando código de derramamento deve ser emitido.

Dadas estas restrições, os algoritmos de alocação interprocedimental executam o trabalho em dois níveis. A partir do grafo de chamadas estático o alocador interprocedimental altera convenientemente a convenção de registradores voláteis/preservados para cada procedimento, de modo a impedir a reutilização de alguns e priorizar o uso de outros registradores. A seguir um alocador intraprocedimental que conhece as convenções calculadas é aplicado a cada procedimento. Todos os algoritmos têm a mesma idéia geral: *dois procedimentos que podem estar ativos simultaneamente não devem utilizar os mesmos registradores, pois assim evitam o custo de gerenciamento do contexto durante chamadas.* Desta observação conclui-se que um alocador interprocedimental tende a ser mais efetivo em programas cujo grafo de chamadas estático é mais largo que profundo, pois estes são os casos onde um grau maior de reutilização de registradores pode ser alcançado.

Como se verá, a abordagem em dois níveis simplifica o tratamento de recursividade e chamadas indiretas e ao mesmo tempo evita tratar o número excessivo de arestas em um grafo de interferência global.

5.3.1 Alocação em Tempo de Ligação

A aparente necessidade de construir o grafo de chamada estático fez com que Wall ([Wal86]) desenvolvesse uma técnica de alocação interprocedimental executada na fase de ligação.[1] O algoritmo trabalha sobre informações produzidas durante a geração de código. Basicamente o gerador de código foi instrumentado para:

1. Fazer otimizações convencionais dentro de cada procedimento;
2. Gerar código objeto (correto) considerando todas as variáveis em memória;

[1] do inglês: *link time*.

3. Anotar o código com ações a serem tomadas se o alocador for executado em tempo de ligação.

Para a simples atribuição ($c \leftarrow a + b$) o código gerado é semelhante àquele da Figura 5.2:

INSTRUÇÕES	AÇÕES
`ld [ "a" ], r1`	REMOVE.a
`ld [ "b" ], r2`	REMOVE.b
`add r1, r2, r3`	OP.a, OP.b, RESULT.c
`st r3, [ "c" ]`	REMOVE.c

Figura 5.2: Código anotado para alocação em tempo de ligação.

Se as variáveis a b e c forem promovidas a registradores, REMOVE.a REMOVE.b e REMOVE.c indicam que a primeira a segunda e a última instrução podem ser removidas. OP.a indica que r1 deve ser trocado pelo registrador associado à variável a. O mesmo vale para RESULT.c.

Detalhes sobre anotação do código não serão mais discutidos. Supõe-se que é possível definir ações convenientes que permitem "reescrever" o código após a alocação de registradores e relocação de endereços, durante o processo de ligação.

O Algoritmo Básico

Idealmente a alocação consiste em promover o maior número de variáveis do programa aos registradores disponíveis ao alocador. A idéia por trás do algoritmo é que variáveis locais de procedimentos não simultaneamente ativos podem ser agrupadas e associadas ao mesmo registrador.

A geração dos agrupamentos é feita percorrendo um grafo de chamadas acíclico (recursão é tratada à parte), associando as l variáveis locais dos procedimentos *folha* a *grupos*, iniciando de zero até $l-1$; as m variáveis locais de procedimentos não *folha* são associados aos grupos $(G+1, G+2, \ldots, G+m)$, onde G é o maior grupo ocupado por uma variável local de um filho no grafo. Cada variável global é colocada em um grupo diferente, visto que elas podem estar vivas em mais de um procedimento. Esta construção gera grupos contendo variáveis que nunca interferem entre si, e que por esta razão podem receber o mesmo registrador.

A primeira coluna da Figura 5.3 apresenta as variáveis de cada um dos procedimentos da Figura 5.1. A aplicação do algoritmo acima gera os grupos indicados na segunda coluna.

$$\begin{array}{ll} A = \{a_1\} & G_0 = \{b_1, c_1\} \\ B = \{b_1, b_2\} & G_1 = \{b_2, c_2\} \\ C = \{c_1, c_2, c_3\} & G_2 = \{c_3\} \\ & G_3 = \{a_1\} \end{array}$$

Figura 5.3: Variáveis e grupos para um programa simples.

O alocador escolhe os grupos mais freqüentemente utilizados e associa os registradores físicos a eles. A freqüência de uso de um grupo g é dada pela somatória das freqüências de uso das variáveis pertencentes ao grupo; a freqüência de uso de uma variável v é o produto da freqüência de chamada do procedimento P onde ela ocorre, pela estimativa de referências locais à variável v ao longo de P (referências em malhas com peso 10).

Uma maneira natural de estimar a freqüência de chamadas a um procedimento P é multiplicar a freqüência de cada chamador C pelo número estimado de chamadas a P ocorridas dentro de C. Wall encontrou que estas suposições tendem a gerar resultados normalmente absurdos e, por isso, resolveu estimar a freqüência de chamadas a um procedimento P como sendo o somatório das freqüências de seus chamadores.

Alocar apenas as variáveis mais usadas conforme as heurísticas acima pode gerar situações em que duas variáveis no mesmo procedimento nunca estão vivas simultaneamente, mas apenas uma delas recebe registrador. Tentando eliminar essas anomalias, um grafo de interferência local é construído durante a geração de código e as duas variáveis passam a ser consideradas uma só. Da mesma forma um pequeno número de registradores é destinado à emissão de código que dispensa a execução do alocador. Com esses refinamentos, o algoritmo que é global ganha também "inteligência" local.

Recursividade e Chamadas Indiretas

Chamadas a procedimentos que geram circularidades no grafo de chamadas estático podem potencialmente destruir todos os registradores vivos a partir do procedimento chamado. Para criar novas instâncias dos registradores o algoritmo faz o gerenciamento do contexto pertencente ao componente fortemente conexo no instante das chamadas que definem arestas de retorno.[2]

A Figura 5.4 mostra um grafo de chamadas estático para um programa com recursão. Os rótulos nos vértices representam os registradores utilizados no procedimento relativo ao vértice. A chamada de A em D determina uma aresta de retorno; portanto, imediatamente antes (depois) desta chamada, o alocador insere código para armazenar (restaurar) o conteúdo de $(r1 - r7)$.

Identificadas as arestas de retorno, o grafo de chamadas pode ser feito acíclico, e o algoritmo básico aplicado.

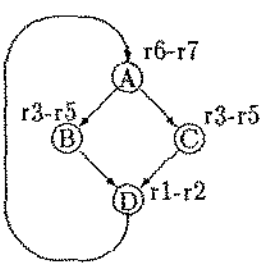


Figura 5.4: Um grafo de chamadas recursivo e registradores usados.

Desde que uma chamada indireta não é representada no grafo de chamadas, ela não é considerada no cálculo dos grupos, exigindo também tratamento diferenciado.

[2]Para evitar análise de fluxo em tempo de ligação todos os registradores usados no componente são considerados vivos.

A solução para recursividade também se aplica a esta situação. O alocador armazena o contexto sempre que encontra uma chamada indireta. No grafo da Figura 5.5, quando o alocador encontra uma chamada indireta dentro de P (possivelmente a Q), inclui código para armazenar (restaurar) os registradores ($r6 - r15$).

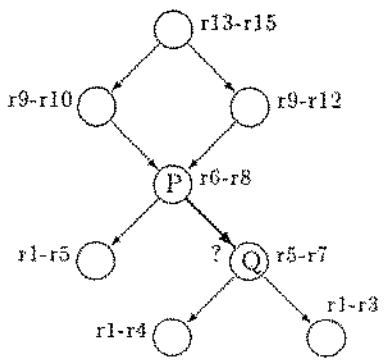


Figura 5.5: Grafo com chamada indireta.

Comentários

O alocador de Wall foi testado com vários *front-ends* tornando possível verificar a sua viabilidade em FORTRAN, Modula-2 e C. Os resultados obtidos foram significativos. Em alguns programas o tempo de execução foi reduzido em até 28% para uma máquina RISC com 52 registradores disponíveis ao alocador. O número de referências à memória, devido a variáveis escalares, foi reduzido em até 99% [Wal88]. As vantagens decrescem com o aumento do número de variáveis e procedimentos.

Em verdade, o algoritmo aqui mencionado pode ser melhorado. No momento em que um grupo é escolhido para receber um registrador físico, todas as variáveis pertencentes ao grupo são automaticamente associadas àquele registrador. Vistas por outro ângulo, as variáveis de um procedimento que foram promovidas a registrador não necessariamente são as mais utilizadas localmente. Para fazer melhor uso dos registradores, sugere-se que quando um grupo G contendo uma variável qualquer v do procedimento P é promovido a registrador, a variável v' de P, que efetivamente receberá o registrador, será a mais utilizada localmente que ainda está em memória. Observe-se que isto não muda o processo de alocação, mas apenas a associação variável/registrador.

A principal vantagem desta técnica é a sua adequação aos ambientes de desenvolvimento disponíveis. A exemplo de janelas deslizantes seu uso é totalmente transparente ao programador.

5.3.2 Alocação Interprocedimental Cooperativa

Analisando um conjunto de programas Lisp, Steenkiste e Hennessy [SH89] observaram que a utilização de algoritmos robustos de alocação intraprocedimental não afetou significativamente o tempo de execução desses programas. Isto resulta dos seguintes dados por eles encontrados:

- Para uma arquitetura particular, MIPS-X [GHPR88], apenas 11 instruções, em média, são executadas entre duas instruções sucessivas de chamada ou retorno de procedimento. Nos

SPEC *benchmarks* escritos em C, este número é 30; para os escritos em FORTRAN, 80; [CKDK91];

- O número de registradores necessários por procedimento é muito pequeno, inferior a 5, em média.

Em um ambiente com estas características, um alocador interprocedimental é muito importante.

A técnica desenvolvida é, em princípio, restrita a programas de um único módulo, mas aplicável a qualquer linguagem. Em um primeiro estágio, o otimizador faz todas as transformações sobre o código gerado, a seguir o grafo de chamadas estático é construído e a alocação aplicada.

O Algoritmo Básico

Um outro dado importante obtido por Steenkiste e Hennessy foi que 90% dos procedimentos executados dinamicamente têm altura menor do que ou igual a três no grafo de chamadas estático (procedimentos folha têm altura zero, aqueles que os chamam um e assim por diante). De posse deste dado, os procedimentos mais próximos às folhas foram priorizados durante a alocação interprocedimental.

Partindo das folhas, o algoritmo interprocedimental aplica uma técnica intraprocedimental a cada procedimento, mas inibe a reutilização de determinados registradores. A alocação de um procedimento não folha P só tem início quando todos os seus descendentes já foram processados. Para fazer alocação em P, o alocador interprocedimental não permite o uso de $r1$-r_k, onde r_k é o mais alto registrador associado a qualquer das variáveis de um sucessor de P no grafo de procedimentos.

Em um esquema assim é possível que após atingida certa altura no grafo de chamadas estático, todos os registradores tenham sido utilizados e alguns procedimentos sequer foram submetidos ao alocador.

Uma solução simples para este problema é armazenar todos os registradores sempre que um procedimento requer mais registradores que o número ainda disponível. Considerando que os registradores seriam gerenciados em alguns procedimentos, do ponto de vista do alocador interprocedimental, eles voltam a estar disponíveis para alocação. Em outras palavras, os procedimentos onde gerenciamentos ocorrem não propagam usos de registradores no grafo de chamadas estático, comportando-se como folhas. A Figura 5.6 mostra uma alocação seguindo este esquema (oito registradores disponíveis). O vértice marcado indica onde o contexto foi gerenciado.

Esta solução tem a desvantagem de tornar alguns procedimentos críticos para efeito de gerenciamento. Se um destes for mais executado que seus sucessores, o tempo de execução do programa pode até ser aumentado. Por isto os autores usaram um esquema *cooperativo* de alocação de registradores. Sempre que o número de registradores for exaurido, o alocador interprocedimental "instrui" o alocador intraprocedimental a considerar todos os registradores como voláteis, forçando gerenciamentos de contexto (quando chamadas são cruzadas) em todos os procedimentos que precedem, no grafo de chamadas estático, o procedimento onde a exaustão foi verificada.

Recursividade e Chamadas Indiretas

O tratamento de chamadas indiretas é basicamente o mesmo proposto por Wall, com pequenas mudanças motivadas pela estrutura do compilador onde o alocador foi implementado.

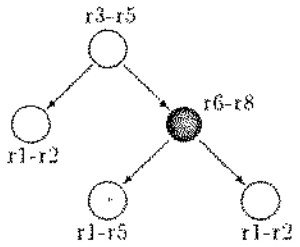


Figura 5.6: Propagação de registradores.

Recursividade tem um tratamento diferente. Para eliminar ciclos no grafo de chamadas estático, cada componente fortemente conexo [ASU86] é reduzido a um único vértice. Dentro de um destes componentes a alocação é feita em um estilo intraprocedimental, com todos os registradores sendo usados na convenção volátil. A Figura 5.7 é uma comparação com o método de Wall. Os rótulos nas arestas indicam o gerenciamento necessário no momento da respectiva chamada. A técnica de Wall faz o gerenciamento total quando da chamada recursiva. A presente técnica armazena "parcialmente" os registradores no momento de qualquer chamada dentro do componente fortemente conexo (mostrado na Figura).

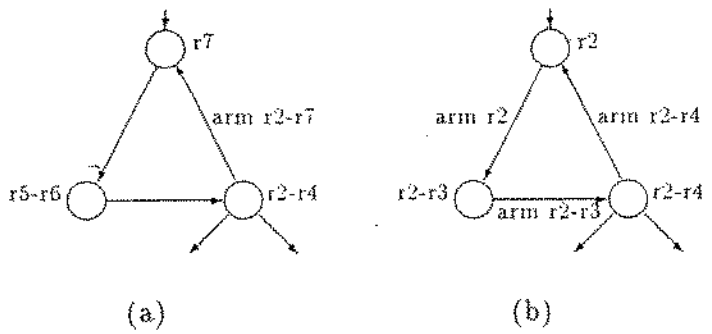


Figura 5.7: Tratamento de recursão: (a) Wall; (b) Steenkiste.

Com essa decisão, o uso propagado para alocação interprocedimental por um componente fortemente conexo é determinado pelo maior usuário de registradores dentro deste. Observe, entretanto, que este componente ainda deve seguir o critério ascendente, só podendo ser processado após seus sucessores externos (ao componente).

Comentários

A efetividade deste método é função do tempo gasto pelo programa executando procedimentos próximos às folhas, onde nenhum gerenciamento é necessário.

O tratamento de recursão é fortemente influenciado pela linguagem Lisp. [SH89] observa que mesmo após a remoção de recursões triviais, 22% das chamadas em Lisp continuam recursivas e,

uma vez que nestes casos o contexto deve ser armazenado, sua abordagem é mais razoável. Para Modula-2 e C, [Wal88] encontrou que o comportamento de programas recursivos é diferente. Nestas linguagens grande número de chamadas ocorre dentro de cada componente fortemente conexo.

A vantagem da técnica, pelo menos em relação à de Wall, é que nenhuma realocação é necessária para procedimentos em bibliotecas, bastando que a informação de uso dos registradores esteja disponível. Entretanto, nenhum tratamento é feito com relação a variáveis globais e compilação de módulos separados.

5.3.3 Alocação Interprocedimental em Um Passo

Chow [Cho88] apresenta um aprimoramento da técnica descrita na seção anterior. O método de alocação é muito semelhante e também segue um esquema ascendente, com um procedimento sendo processado por vez e seguindo uma ordenação topológica. A diferença é que o algoritmo é capaz de tratar registradores seguindo as convenções: volátil e preservado. Uma convenção *default* dos registradores é utilizada para estabelecer uma "interface" quando possíveis arestas não estão representadas no grafo, por exemplo, em compilação de módulos separados ou chamadas indiretas. Uma outra forma de ver isto é dividir o grafo de chamadas estático em subgrafos, cada um deles com total informação sobre chamadas, e aplicar a alocação interprocedimental nos subgrafos. Nos vértices de *conexão* entre os subgrafos a convenção *default* é respeitada.

A exemplo da alocação cooperativa, o objetivo é impedir a reutilização de registradores voláteis nos ascendentes de procedimentos que os empregam. Note-se, portanto, que em princípio, um benefício maior é alcançado quando todos os registradores seguem a convenção volátil.

O Algoritmo Básico

O esquema interprocedimental foi implementado tendo por base o algoritmo intraprocedimental do próprio Chow, descrito no capítulo 4. Quando a alocação interprocedimental está em uso, todos registradores são vistos como voláteis.

Com a informação interprocedimental, o alocador intraprocedimental pode calcular a prioridade de alocação de cada variável em relação a cada um dos registradores, considerando o fato da variável cruzar ou não chamadas e que registradores são utilizados em cada procedimento eventualmente cruzado. O registrador com maior prioridade (*LUCRO*) é, obviamente, o escolhido. Isto é uma generalização da abordagem na página 57.

Por exemplo, considere a alocação de registradores para o procedimento raiz do grafo de chamadas estático da Figura 5.8. Suponha que R possui duas variáveis: v e v' (interferindo entre si localmente); v cruza uma chamada a P e v' cruza chamadas a P e Q. Quando calculando a prioridade de v, os registradores $r1$-$r6$ são penalizados, mas não $r7$, pois v não "interfere" com o procedimento Q. Para v' resta associar $r8$, o único registrador não penalizado no cálculo da prioridade.

Recursividade e Chamadas Indiretas

O tratamento de recursão e chamadas indiretas é uniforme: procedimentos alvo de recursão ou indireção devem manter como "interface" externa a convenção *default*. Isto implica que todos os registradores considerados como preservados pela interface e utilizados em um subgrafo, cuja raiz (R) se enquadra na situação acima, devem ser gerenciados em R. Em contrapartida, todos os

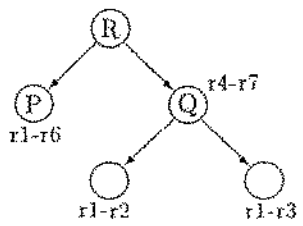


Figura 5.8: Distribuição de registradores em um programa.

procedimentos que são fontes de arestas de retorno, ou fazem chamadas indiretas, devem supor que os registradores voláteis da convenção *default* serão corrompidos, gerenciando algum contexto se necessário.

Suponha que na Figura 5.5, a convenção *default* trata $r5$-$r10$ como preservados e os demais registradores como voláteis. Resultaria que antes de P chamar Q, deveria armazenar $r11$-$r15$; no prólogo de Q, $r5$-$r7$ deveriam ser armazenados.

Este mesmo método é adequado para permitir compilação de módulos separados. Tudo que um procedimento chamando outro não pertencente ao módulo sendo compilado deve supor é a convenção *default*. Igualmente, todo procedimento que pode ser chamado de outro módulo deve respeitar a mesma convenção.

Comentários

Além de fazer melhor uso da informação interprocedimental que a técnica anterior, este método dá um tratamento mais sistemático para compilação de módulos separadamente, cada um submetido à alocação interprocedimental com uma convenção *default* única.

Chow utilizou alguns *benchmarks* para verificar a eficácia do seu alocador interprocedimental. Os programas de teste foram escritos em C e Pascal e são basicamente utilitários UNIX. A máquina alvo era RISC (MIPS), 20 registradores foram disponíveis ao alocador interprocedimental e a convenção *default* foi: 11 voláteis e 9 preservados.

Seus experimentos tiveram o mérito de verificar qual o efeito da aplicação de uma técnica interprocedimental sobre a aplicação de um bom algoritmo intraprocedimental. Os resultados mostraram que, se poucos registradores estão disponíveis ao alocador interprocedimental, os resultados de sua aplicação são poucos expressivos. Nos testes, o número de LOAD/STORE devido a variáveis escalares foi reduzido entre zero e 12,5%, mas o tempo de execução melhorou apenas marginalmente, chegando a ser pior em alguns casos (possível conseqüência do gerenciamento concentrado em procedimentos muito executados).

5.3.4 Alocação Interprocedimental em Duas Fases

As duas últimas técnicas apresentadas supõem implicitamente que procedimentos próximos às folhas são mais freqüentemente executados. Com um pequeno número de registradores físicos disponíveis,

esses métodos podem ser pouco efetivos em programas cujo tempo de execução é dominado por procedimentos nas partes mais altas do grafo de chamadas estático.

Dado um grafo de chamadas, a técnica descrita em [SO90], tem por objetivo identificar subgrafos onde a atividade de chamadas é alta. Restringindo a alocação interprocedimental a essas regiões é possível fazer o gerenciamento do contexto de registradores preservados ao longo de toda a região apenas no procedimento de entrada, permitindo o livre uso nos demais. Aparentemente este mecanismo tem a vantagem de não priorizar uma determinada classe de programas, embora os resultados sejam dependentes diretamente da capacidade do algoritmo identificar precisamente as tais regiões.

A Figura 5.9 apresenta o ambiente onde a otimização é executada. A primeira fase do compilador gera para cada arquivo fonte (SRC) dois outros arquivos: o primeiro (INT) referente ao programa expresso em uma linguagem intermediária; o segundo (SUM), chamado arquivo de sumário, contém informações sobre cada procedimento, tais como: freqüência estimada de chamadas a outros procedimentos, procedimentos indiretos e estimativas do número de registradores necessários.

A partir dos arquivos de sumário, o ANALYZER é executado para construir o grafo de chamadas estático e gerar heurísticas e diretivas que guiam a segunda fase do compilador. As heurísticas são armazenadas no D_BASE.

O primeiro requisito para a alocação é determinar as regiões onde o programa executa mais chamadas, doravante denominadas *clusters*. Em otimização, o conceito mais próximo de *clusters* é o de *intervalos* [ASU86]. Intervalos têm sido usados na implementação eficiente de análise de fluxo de controle e são associados com determinados vértices do grafo de controle, denominados raízes. A construção de um intervalo $I(R)$, associado à raiz R, segue três regras:

1. R está em $I(R)$;
2. se todos os predecessores de algum vértice v estão em $I(R)$, então v deve ser adicionado a $I(R)$;
3. os únicos vértices de $I(R)$ são os determinados pelas regras anteriores.

A divisão de um grafo de controle em intervalos se dá calculando $I(R)$ para o vértice inicial[3] e repetitivamente para os não incluídos até então, mas que já tiveram pelos menos um predecessor incluído. Estes certamente são raízes.

Clusters são construídos, sobre o grafo de chamadas estático, de maneira semelhante. A exceção é que os vértices (procedimentos) que serão raízes são previamente escolhidos. Os autores, no entanto, não definem precisamente os critérios de escolha destes vértices. Observe-se que os vértices raízes têm a propriedade desejável de dominar[4] os demais vértices no *cluster*, criando condições para o gerenciamento centralizado.

Sob a condição de raízes pré-determinadas, as regras para escolha dos vértices pertencentes ao *cluster* C, de raiz R, são:

1. R está em C;
2. se todos os predecessores de algum vértice v estão em C e v não foi escolhido para raiz de outro *cluster*, então v deve ser adicionado a C;

[3] Supõe-se existir um.

[4] um vértice v_i domina um vértice v_j se todo caminho chegando a v_j e partindo das raízes do grafo, inclui v_i.

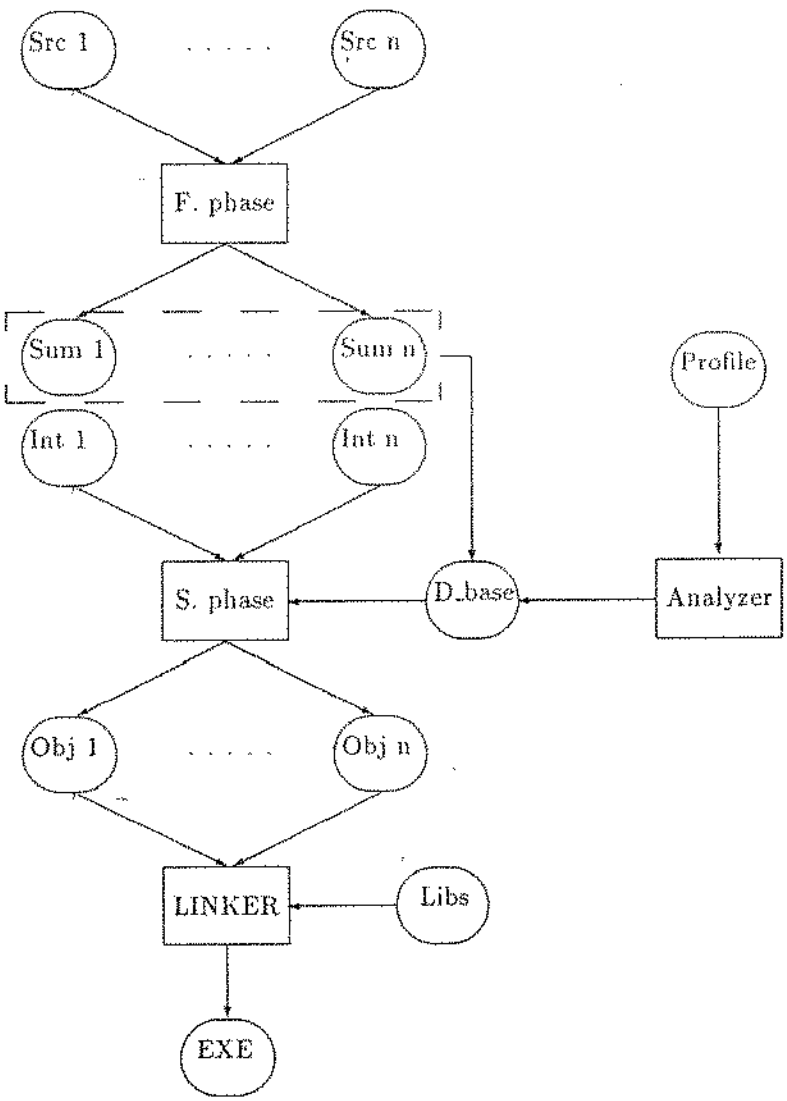


Figura 5.9: Ambiente para alocação interprocedimental.

3. os únicos vértices de C são os determinados pelas regras anteriores.

A primeira fase termina com o armazenamento no D_BASE dos seguintes três conjuntos de registradores para cada procedimento P. Os conjuntos são basicamente os resultados da alocação interprocedimental:

- LIVRES[P] - Registradores que P pode utilizar sem qualquer compromisso de gerenciamento. Eles podem estar vivos durante chamadas dentro de P; não há necessidade também de armazená-los na entrada, nem restaurá-los na saída de P;
- VOLAT[P] - Registradores que seguem a convenção normal volátil;
- PRESERV[P] - Registradores que seguem a convenção normal preservado;

O Algoritmo Básico

Determinados os *clusters*, o algoritmo interprocedimental propaga as informações e gera os três conjuntos acima definidos. Este processo, na sua totalidade, é relativamente complexo, de modo que apenas uma visão geral é dada.

Como alocação interprocedimental é restrita ao interior de cada *cluster*, a raiz permanece tendo o compromisso de manter alguma convenção intraprocedimental (ou *default* na nomenclatura usada na subseção anterior). O mesmo vale para procedimentos no *cluster* que executam chamadas a outros procedimentos não pertencentes ao *cluster*. Estes procedimentos devem gerenciar os registradores voláteis; a raiz os preservados.

O objetivo passa a ser distribuir os registradores preservados aos procedimentos, dando-lhes o direito de não fazer qualquer gerenciamento, ou seja, estes registradores são divididos entre os conjuntos LIVRES dos procedimentos no *cluster*. A distribuição baseia-se em prioridades heurísticas determinadas durante a primeira fase e armazenadas em D_BASE. Neste processo, a exemplo das demais técnicas, procedimentos nunca vivos simultaneamente podem receber os mesmos registradores.

Os dois outros conjuntos, VOLAT e PRESERV, são definidos respeitando as características dos LIVRES nos demais procedimentos. Os conjuntos VOLAT dos procedimentos que fazem chamadas externas ao *cluster* são iniciados com os registradores voláteis para respeitar a convenção intraprocedimental. O conjunto PRESERV da raiz é iniciado como vazio porque os registradores preservados serão destinados à distribuição como livres dentro do *cluster*. Seu comportamento externo é garantido gerenciando na sua entrada (e saída) os conjuntos LIVRES de todos os procedimentos do *cluster*.

Os conjuntos VOLAT e PRESERV são aumentados em função da distribuição dos LIVRES. Mais especificamente, dado um procedimento P que tem r_i em seu conjunto LIVRES todos os seus sucessores ($Suc[P]$) devem associar r_i aos seus PRESERV, pois P não tem o compromisso de gerenciar o valor de r_i quando cruza chamadas. Por outro lado, os predecessores de P ($Pred[P]$) devem associar r_i aos conjuntos VOLAT, pois P não necessariamente manterá o valor que r_i durante sua execução.

Precisamente, os conjuntos VOLAT[P] e PRESERV[P] podem ser calculados resolvendo as seguintes equações:

$$\text{VOLAT}[P] = \begin{cases} \text{VOLÁTEIS} & \text{se } P \in \text{FOLHAS}(C) \\ \bigcup_{Q \in Suc[P]} (\text{LIVRES}(Q) \cup \text{VOLAT}(Q)) & \text{caso contrário} \end{cases}$$

$$\text{PRESERV}[P] = \begin{cases} \emptyset & \text{se } P = \text{RAIZ}(C) \\ \bigcup_{Q \in Pred[P]} (\text{LIVRES}(Q) \cup \text{PRESERV}(Q)) & \text{caso contrário} \end{cases}$$

Onde, FOLHAS(C) é o conjunto de todos os vértices no *cluster* C que não têm sucessores, ou que se têm, nenhum deles pertence a C. VOLÁTEIS são os registradores tratados como voláteis na convenção *default*.

Na segunda fase do compilador, o alocador intraprocedimental pode ser facilmente modificado para trabalhar com os conjuntos. O conjunto VOLAT pode ser visto como os registradores voláteis. As variáveis que devem receber registradores preservados têm o conjunto LIVRES inspecionado antes de PRESERV, pois assim gerenciamentos podem ser evitados.

Cada raiz de *cluster* tem a incumbência adicional de armazenar todos os registradores livres usados no cluster. (Estes correspondem exatamente à união dos conjuntos VOLAT e LIVRES da raiz reduzidos aqueles que pela convenção padrão são voláteis).

Recursividade e Chamadas Indiretas

O tratamento de chamadas recursivas é imediato. Todo procedimento alvo de uma aresta de retorno é feito ser raiz de um *cluster* e, portanto, todo o contexto de registradores preservados é gerenciado.

Para chamadas indiretas o método é bastante conservativo: Arestas são introduzidas no grafo de chamadas estático ligando todo procedimento que faz chamada(s) indiretas a cada um dos que potencialmente podem ser chamados indiretamente. Assim a propagação do uso de registradores é automática e o algoritmo básico pode ser aplicado.

Comentários

A principal observação a respeito da técnica é a necessidade de identificar bem as raízes dos *clusters*, visto que, em função de concentrarem o gerenciamento de um grupo de procedimentos, elas tem grande influência nos resultados dos programas compilados usando este método. Em alguns programas de teste, os autores encontraram que *clusters* têm tamanho médio de apenas três procedimentos.

Em uma máquina com 32 registradores (metade são preservados) os resultados observados são modestos. Nos seis programas, apenas três apresentaram redução no tempo de execução, apenas um deles significativo com 6%. Os números foram obtidos em um simulador que ignora efeitos do *cache*; na prática, os resultados são inconclusivos.

5.4 Alocação em *hardware* ou *software*?

Conjuntos de registradores organizados na forma de janelas deslizantes têm sido razão de muitos estudos e controvérsias desde o surgimento das primeiras máquinas RISC (nenhuma máquina CISC

implementa o mecanismo). Seus defensores vêem uma série de benefícios [PW89]:

- O sistema garante a reutilização dos registradores automaticamente quando possível, ao mesmo tempo, associa registradores diferentes quando necessário;
- O sistema de tratamento de estouros (*underflow/overflow*) é transparente ao *software* do usuário;
- Um maior número de registradores pode estar (geralmente está) disponível sem requerer mais *bits* nas instruções;
- Experimentos comprovam que a cadeia dinâmica de chamadas segue um princípio de localidade [TS83]; em outras palavras, não há grandes seqüências de chamadas sem retornos, e vice-versa. Portanto o acesso às janelas tem comportamento similar a acessos ao *cache*. Esta semelhança garante que estouros são relativamente raros.

Os críticos da abordagem argumentam que janelas têm os seguintes inconvenientes:

- É difícil determinar o número ideal de registradores por janela. Mesmo sendo o número médio de registradores necessários por procedimento baixo, existem anomalias. Como a flexibilidade é pequena, um superdimensionamento é feito;
- O custo de tratar estouros é alto. Muitos armazenamentos desnecessários podem ser executados, mesmo estando esta informação disponível a compiladores que fazem análise de fluxo de dados.
- A rigidez do mecanismo é pouco sensível à alocação dirigida por *profile*;
- É difícil quantificar o efeito de janelas na implementação de uma arquitetura, embora em arquiteturas, como SPARC, o tempo de ciclo seja determinado pelo acesso ao *cache*;
- O sistema operacional gasta mais tempo na troca de contexto dos processos mesmo quando poucos registradores estão realmente em uso por determinado programa (ou mais precisamente processo). Isto é porque todos os registradores são armazenados/restaurados durante a troca do processo corrente.

As três principais questões a cerca de janelas são: Qual o custo de tratar estouros? Qual o número de janelas e quantos registradores são necessários por janela? Qual o efeito de implementar janelas no tempo de ciclo da máquina?

A primeira pergunta foi respondida fazendo uma análise dos SPEC *benchmarks* executados na SPARC. O GCC foi instrumentado para computar o número de estouros, supondo sete janelas (uma das quais dedicadas ao sistema operacional, vide seção 3.3.2) disponíveis. Os resultados são sintetizados na Tabela 5.1. Os números são ligeiramente aproximados.

Considerando ainda que o número de ciclos gastos (supondo janelas gerenciadas a custo zero) nas instruções chamadas e retorno de procedimentos nos SPEC analisados varia entre 0,6% (eqntott) e 2,5% (li) [CKDK91], o custo de gerenciar janelas é baixo. No 001.gcc-1.35, por exemplo, 1,5% dos ciclos são gastos com chamadas ou retornos, destes 0.86% geram estouros. Na SPARCstation 1+ o

Benchmark	No. estouros	Chamadas+Retorno	Percentual
001.gcc-1.35	184.470	21.500.000	0,86%
008.espresso	104.000	20.165.000	0,51%
022.li[a]	-	-	-
023.eqntott	1.540	6.720.000	0,02%

[a]O compilador GCC disponível foi incapaz de compilar corretamente este *benchmark*.

Tabela 5.1: Freqüência de *overflow/underflow* de janelas na SPARC.

custo de tratar um estouro é em torno de 130 ciclos;[5] o aumento de ciclos devido a gerenciamento de janelas é, portanto, em torno de 1,5%.

O número de registradores foi analisado por [Wal88] e [SH89], que concluiram ser 128 registradores divididos em oito conjuntos a quantidade mais razoável. Esta divisão é porque ter muitas janelas com poucos registradores pode gerar muito tráfego para a memória decorrente de derramamentos dentro dos procedimentos. Conversamente, poucas janelas geram muitos estouros, ainda que localmente nem todos os registradores tenham sido, de fato, utilizados. Os mesmos experimentos mostraram que para conseguir uma efetividade similar aos alocadores estáticos interprocedimentais, o mecanismo de janelas necessita de, no mínimo, duas vezes mais registradores.

O melhor alocador interprocedimental deve ser eficiente onde o programa executa a maior parte do tempo. Janelas têm vantagem neste ponto, pois a sua natureza dinâmica é muito apropriada a isto. Em tempo de compilação é difícil fazer julgamentos precisos e o grafo de chamadas estático é muito conservativo em relação às chamadas que ocorrem dinamicamente.

Algoritmos mais recentes [SO90] trabalham em estimar os "gargalos" do programa, implicitamente modelando a localidade de chamadas, mas os resultados ainda são pouco expressivos. Estatísticas de execuções prévias do programa submetido ao alocador também são importantes para aumentar a eficiência dos algoritmos estáticos. O algoritmo de Wall, quando trabalhando com estas informações, originou resultados melhores que o esquema de janelas. Isto se verificou até mesmo em programas de tamanhos consideráveis [Wal88].

5.5 Conclusões

O grau de pesquisa na área, as tendências tecnológicas e os ambientes onde os métodos aqui descritos podem ser aplicados permitem tirar algumas lições a respeito dessa otimização:

Primeiro, é muito difícil gerenciar manualmente mudanças no código fonte de grandes sistemas, mesmo sem fazer otimizações interprocedimentais. Enquanto ambientes de gerenciamento automático do desenvolvimento não estiverem disponíveis operacionalmente na maioria das instalações, é pouco provável que alocação interprocedimental venha a ser utilizada em grande escala.

Segundo, alocação interprocedimental é, de certo modo, dependente da linguagem fonte. Para

[5]Este dado foi obtido empiricamente e é fortemente dependente do desempenho do subsistema de memória. Os custos de tratar *overflow* e *underflow* são diferentes, mas como o número deles é o mesmo, pode-se abstrair deste fato.

a linguagem C, ela não parece tão importante (quando poucos registradores estão disponíveis), em comparação com otimizações ambiciosas restritas a procedimentos. Linguagens recentes, por outro lado, têm enfatizado o uso de módulos com interface externa bem definida, propiciando mais informação ao alocador interprocedimental e possivelmente melhores resultados no caso de compilação em separado. Na mesma linha, o advento de linguagens orientadas a objetos [TL90, Irl91] tende a aumentar o percentual de chamadas de procedimentos e, conseqüentemente a necessidade deste tipo de otimização. Em grandes conjuntos de registradores ela passa a ser uma necessidade.

Finalmente, alocação interprocedimental em *software* traz ainda mais complicações aos compiladores e é cara. Pelo menos por enquanto, sua aplicação será restrita a programas onde o tempo de execução é crítico, justificando o custo das otimizações. Neste sentido, janelas deslizantes se apresentam como um atalho.

Capítulo 6

Reorganização de Instruções

"The most valuable of all talents is never to use two words when one will do."

Thomas Jefferson

6.1 Introdução

O processamento simultâneo de instruções sucessivas em um processador *pipelined* pode resultar na situação onde uma instrução faz referência a um dado ainda não computado por alguma outra instrução precedente. Existe, portanto, a necessidade de atrasar a instrução alvo até que o operando tenha sido, de fato, calculado. Em geral, arquiteturas identificam e solucionam a situação em *hardware*. Hennessy [Hen84] estimou em 15% o aumento no tempo de ciclo básico para suportar um mecanismo desta natureza. Algumas arquiteturas RISC [GHPR88] tratam estas interdependências, ou *interlocks*, entre instruções através do uso explícito de instruções nop, ao custo de penalidades no tamanho do código.

O tratamento de *interlocks*, em *hardware* ou via nop, aumenta o tempo de execução dos programas. Uma alternativa é identificar, em tempo de compilação, os *interlocks* e reorganizar (ou reescalonar) o código, de modo a minimizar as interdependências entre instruções, proporcionando um menor tempo de execução. Claramente, instruções não podem ser arbitrariamente reorganizadas; em qualquer reorganização a semântica do programa deve ser preservada. Dentro de um bloco básico, por exemplo, um grafo acíclico orientado (DAG) expressa os relacionamentos entre instruções que avaliam expressões; qualquer reordenação deve manter tais relacionamentos, embora as seqüências parciais possam ser diferentes.

Este capítulo trata de técnicas de reorganização de instruções. A próxima seção estabelece algumas propriedades do problema, com ênfase no processo restrito a blocos básicos; os algoritmos para escalonamento são abordados na seção 6.3; inclui-se nesta uma técnica desenvolvida no decorrer deste trabalho. Para técnicas cujo escopo é maior que um bloco básico dedica-se a seção 6.4. A interação entre reorganização e alocação de registradores está na seção 6.5. A seção 6.6 conclui.

6.2 Propriedades da Reorganização

Complexidade

Conforme discutido na seção 4.2, árvores de expressões podem ter código ótimo[1] gerado eficientemente para máquinas não *pipelined*. Recentemente Proebsting e Fischer [PF91] propuseram um algoritmo similar para geração de código ótimo em máquinas RISC *pipelined* e *delayed-load* (aquelas onde a instrução de LOAD tem latência e as aritméticas não). O número de registradores utilizados excede a quantidade necessária em máquinas não *pipelined* exatamente no número de ciclos de latência do LOAD. O número mínimo de ciclos é garantido.

Em termos gerais, o problema de gerar código ótimo para DAGs em máquinas *pipelined* pode ser visto como aquele de emitir código tal que o custo de uma instrução é dependente das instruções que a seguem ou precedem. É óbvio, portanto, que este problema é, no mínimo, tão difícil quanto o de geração de código ótimo para máquinas com instruções de custo fixo e, por isto, também é *NP-completo*.

Normalmente a geração de código ignora a presença de *pipeline*. Posteriormente a ordem das instruções emitidas é modificada visando a minimização dos *interlocks*. Neste esquema não há mudança no número de instruções (a menos de nops), mas possivelmente no número de ciclos para executá-las. Se o código for sub-ótimo (com respeito ao número de instruções), os algoritmos de reorganização, mesmo removendo todos *interlocks*, não o deixam ótimo. Hennessy e Gross [HG83] provaram que apenas minimizar o número de *interlocks* em um bloco básico é um problema *NP-completo*, se a separação requerida entre instruções é maior que um ciclo.

Onde executar

Existem duas abordagens para fazer o escalonamento de instruções: antes ou após a emissão do código objeto. No primeiro caso, o reorganizador trabalha sobre a linguagem intermediária, mas deve de alguma forma conhecer as instruções que serão emitidas. Em geral, numa máquina RISC, é factível executar a seleção do código sem conhecer efetivamente quais os registradores físicos envolvidos na computação.

A principal vantagem de fazer reorganização nesse estágio é a ausência de *interlocks* adicionais causados por associações de variáveis distintas ao mesmo registrador. Por outro lado, é possível aumentar o tamanho das cadeias de vida das variáveis, tornando mais difícil fazer a alocação; se derramamento for necessário, então existem os seguintes problemas:

- A maior latência de instruções de acesso à memória (comum em RISCs) faz com que o custo de derramamento normalmente exceda o lucro com a reorganização;[2]
- Dependendo das características da máquina alvo, pode ser necessário executar novamente o reorganizador, desde que uma instrução de acesso à memória imediatamente seguida de outra que utiliza o registrador carregado, normalmente, gera *interlock*;

[1] Otimalidade diz respeito a número de instruções e número de registradores, mas como seleção de código tem sido ignorada isto equivale também ao número mínimo de ciclos.

[2] Observe-se, todavia, que à medida que a latência das demais instruções cresce em comparação às latências de acesso à memória, a situação pode se inverter.

A reorganização sobre o código de montagem tem ainda a vantagem de ser igualmente aplicável a código escrito à mão. A ausência de *interlocks* no *hardware* torna muito difícil escrever programas corretos (e eficientes) em linguagem de montagem.

A Representação do Problema

A estrutura de dados que suporta as restrições para reorganização dentro de um bloco básico é um um grafo de dependência de dados (ou DAG de código), onde cada vértice representa uma instrução e as arestas dependências de dados entre elas. Especificadas as arestas, qualquer classificação topológica [ASU86] do grafo corresponde a uma seqüência que avalia corretamente cada expressão no bloco básico. As raízes do DAG são instruções que não dependem de qualquer outra. Uma aresta (i, j) rotulada com o valor l significa que j não deve ser escalonada antes que l ciclos tenham decorrido desde o escalonamento de i.

Arestas são de três tipos: o primeiro representa uma dependência RAW e é introduzida para evitar usos de operandos (memória ou registrador) antes da sua respectiva definição; dependências WAR impedem a redefinição de um operando antes que todas as as instruções que o utilizam já tenham sido escalonadas; dependências WAW garantem a ordenação correta para código não otimizado.[3] O DAG é normalmente construído percorrendo o bloco básico da última até a primeira instrução, notando cada definição ou uso de um operando e relacionando os usos e/ou definições que necessitam precedê-los.

A Figura 6.1 mostra o DAG de código correspondente à avaliação das raízes de um polinômio de segundo grau. Por simplicidade supõe-se que existe no mínimo uma raiz e que instruções de raiz quadrada, multiplicação e divisão estão presentes na máquina alvo. Todas as instruções têm custo unitário e apenas multiplicação (`mul`) e raiz quadrada (`sqrt`) têm latência: um ciclo. Sob estas condições, a seqüência original tem três *interlocks*: $\{(v_2, v_3), (v_3, v_4), (v_8, v_9)\}$.

Dependências de recursos

Uma definição precisa do problema de reorganização o torna ainda mais complexo. Em função da contenção em determinados recursos da arquitetura alvo, é possível que duas instruções sem qualquer dependência de dados gerem *interlocks* se executadas uma seguindo a outra. Considere, por exemplo, a unidade de ponto flutuante da arquitetura SPARC na página 33. Uma comparação entre registradores de ponto flutuante e posterior verificação da condição requer uma instrução sobre registradores inteiros intercalando-as. Problemas semelhantes acontecem com o Motorola 88000 [Mel89]. Estes tipos de dependência são denominados de dependências estruturais ou de recursos [BHE91], sendo comuns no escalonamento para VLIW ou qualquer tipo de arquitetura horizontal. Ver [DLSM81] para um *survey* de reorganização nesse contexto.

Evitar dependências estruturais exige verificar, em cada instante do escalonamento, as interseções entre os recursos utilizados por cada instrução já escalonada e aqueles ocupados por cada um dos candidatos a escalonamento. Interseções deveriam ser averiguadas em todos os ciclos usados pela instrução candidata. Os algoritmos de escalonamento ignoram, ou tratam parcialmente através de métodos *ad hoc*, dependências estruturais, mesmo que isto acarrete alguma ineficiência.

[3] Rigorosamente não há necessidade de serializar anti-dependências; é suficiente que o algoritmo de reorganização emita todas as instruções que usam um dado, antes que ele seja redefinido. Um DAG assim formado permite uma maior flexibilidade de reorganização, principalmente se há muitas reutilizações de registradores, entretanto o processo se torna bem mais complicado.

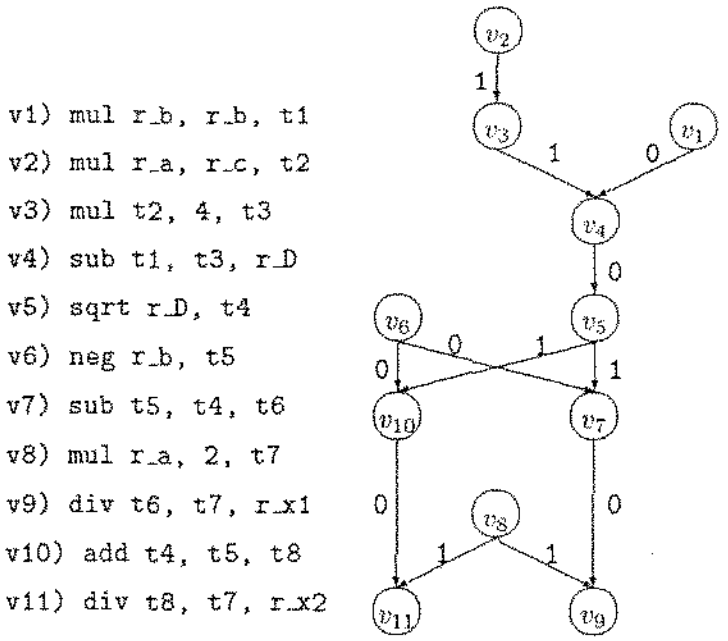


Figura 6.1: Um grafo de dependência de dados para cálculo de raízes de equação de grau 2.

Alguém pode sugerir a modelagem de dependências de recursos através de arestas extras no grafo de dependências de dados. A questão é que face à natureza acíclica do grafo torna-se necessário estabelecer algum critério para a inserção da aresta de sorte a mantê-lo livre de ciclos e não gerar dependências artificiais para a reorganização.

Descobrir a orientação destas arestas é um problema oneroso, mesmo em situações bem simples. Por exemplo, considere a Figura 6.2, onde G' e G'' são subgrafos, tais que todos os vértices de G' precedem v_i que por sua vez precede todos os vértices de G''. Há dependência de recursos entre v_i e v_j. A orientação da aresta entre v_i e v_j que não cria restrições artificiais só pode ser determinada se os escalonamentos de G' e G'' forem conhecidos. Portanto, para determinar a orientação que não cria restrições artificiais ao escalonamento, pode ser necessário fazer a própria reorganização.

6.3 Reorganização em Blocos Básicos

As técnicas heurísticas mais comuns para fazer reorganização de instruções dentro de um bloco básico empregam o conceito de listas de escalonamentos. Em qualquer instante há um conjunto de instruções prontas, constituindo a lista. Uma instrução na lista está pronta se todos os seus ancestrais no grafo de código já foram emitidos e não há dependência de dados entre ela e as demais já escalonadas.

O reorganizador utiliza várias estimativas para decidir qual a instrução a emitir entre as prontas. Uma heurística muito efetiva é priorizar as instruções mais distantes de qualquer folha no DAG.

Nas próximas subseções se discute alguns algoritmos para reorganizar código dentro de blocos

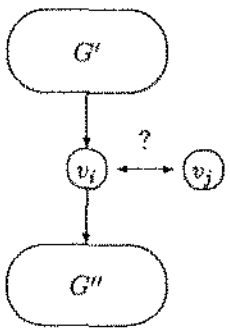


Figura 6.2: Serialização para dependências estruturais.

básicos. O primeiro foi proposto por Bernstein e Gertner em [BG89]. A principal vantagem do algoritmo é a garantia de otimalidade (com respeito ao número de *interlocks*) em situações particulares. Outro algoritmo (empregado em compiladores da SUN) é descrito em Gibbons e Muchnick [GM86]. A terceira técnica apresentada é uma generalização da proposta de Warren [War90] feita durante a elaboração desta dissertação. A técnica de Warren tem sido empregada nos compiladores da IBM para o IBM-RS/6000. Informalmente, antes de cada algoritmo menciona-se o modelo de máquina onde ele é aplicado; optou-se por organizá-los em ordem crescente de generalidade da máquina alvo.

6.3.1 Reorganização Ótima para *Interlocks* de Um ciclo

A proposta de Bernstein e Gertner abrange as situações onde escalonamentos ótimos podem ser conseguido de forma eficiente. A principal restrição do algoritmo é o tamanho dos *interlocks* entre instruções; estes não podem exceder um ciclo. Um outro limitante é que a minimização do número de interdependências está restrita àquelas representadas no DAG, ou seja, o algoritmo não trata diretamente dependências estruturais. Cada instrução tem, supostamente, custo unitário.

O Algoritmo Básico

O algoritmo consiste no estabelecimento prévio de prioridades entre as instruções e no posterior escalonamento em função destas prioridades. O processo de escalonamento é muito simples e consiste de:

1. iniciar o contador de ciclos em zero;
2. repetir até que todas as instruções sejam escalonadas:
   (a) emitir a instrução de maior prioridade que esteja pronta; isto é verificado observando o ciclo em que cada predecessor foi escalonado;
   (b) se nenhuma instrução está pronta (e a máquina não implementa *interlocks* em *hardware*), emita um nop.
   (c) incrementar o contador de ciclos de um;

O problema, é claro, consiste em determinar a lista de prioridades. Para definir precisamente o processo é necessário estabelecer o conceito de ordenação lexicográfica entre seqüências de números naturais, cada uma delas ordenadas em ordem decrescente. Por ordenação lexicográfica, $\{6,2,1\} > \{6,2\} > \{5,4,3,2\}$. Toda seqüência não vazia é maior que $\{\}$.

Define-se também o conjunto de sucessores com *interlock* do vértice v, $S(v)$, como sendo o conjunto de vértices u tais que existe um aresta (v,u) com rótulo 1 no DAG de código.

Seja G o DAG de código com n vértices; atribua a cada vértice v um número de ordem entre 1 e n, $r(v)$, de acordo com as regras que se seguem:

1. escolha um vértice arbitrário v entre as folhas de G e faça $r(v) = 1$;

2. supondo que os números de ordem de 1 a $i-1$ já foram associados. Para cada vértice v com todos os sucessores já processados, defina a seqüência $M(v)$ formada pelos números de ordem associados a cada um dos vértices em $S(v)$ e classificadas em ordem decrescente. Escolha um vértice v' para o qual $M(v')$ é mínimo (por ordenação lexicográfica) e faça $r(v') = i$;

3. repita o passo anterior até que todos os vértices tenham sido rotulados;

4. a prioridade de um vértice é maior, quanto maior o seu número de ordem;

O processo de rotulação ascendente procura capturar a prioridade através da suposição de que um conjunto de vértices está pronto para escalonamento e que a prioridade dos descendentes é conhecida. Como o escalonador, em princípio, segue a ordem imposta pelos rótulos dos descendentes, é simples determinar a prioridade de cada um dos vértices virtualmente prontos. Quanto maior a diferença entre as prioridades de vértices que têm interdependência, mais provável que ela seja removida, por esta razão o segundo passo do algoritmo rotulador escolhe o vértice cujos descendentes estão mais distantes.

Considere, como exemplo, o mesmo grafo de código da Figura 6.1. A aplicação do algoritmo sobre ele está mostrada na Figura 6.3, onde a cada vértice estão associados a seqüência M e o número de ordem que ele recebeu. A Figura 6.3 é resultado do seguinte: primeiro, v_9 e v_{11} têm $M(v_9) = M(v_{11}) = \{\}$. Faça $r(v_9) = 1$. Esteja claro, agora, que a ordem de rotulação não é única. A seguir têm-se $M(v_7) = M(v_{11}) = \{\}$. Seja $r(v_7) = 2$. Como nenhum outro vértice, além de v_{11}, tem todos os sucessores rotulados, então $r(v_{11}) = 3$. Neste ponto, $M(v_{10}) = \{\}$ e $M(v_8) = \{3,1\}$, logo $r(v_{10}) = 4$. Prosseguindo assim, os números de ordem são associados, em ordem crescente de modo a formar a seguinte seqüência: $v_9, v_7, v_{11}, v_{10}, v_6, v_8, v_5, v_4, v_1, v_3, v_2$, onde v_2 e v_9 são os vértices de maior e menor prioridades, respectivamente. A aplicação do algoritmo de escalonamento gera o seguinte escalonamento: $v_2, v_1, v_3, v_8, v_4, v_5, v_6, v_{10}, v_{11}, v_7, v_9$ – que é livre de *interlocks*. Note-se que após o escalonamento da instrução (representada pelo vértice) v_2, a instrução v_3 seria potencialmente escalonada (pois tem maior prioridade), contudo ela não está pronta neste instante, sendo v_1 escolhida.

Um potencial problema do algoritmo é a suposição de que todos os vértices disponíveis ao rotulador em determinado instante seriam vértices prontos em algum momento do escalonamento. Isto nem sempre acontece (exemplo no parágrafo precedente). O que Bernstein provou foi que a seleção do vértice pronto de maior prioridade (regra 2(a) do escalonador), nestes casos, alcança o critério de otimalidade desejado.

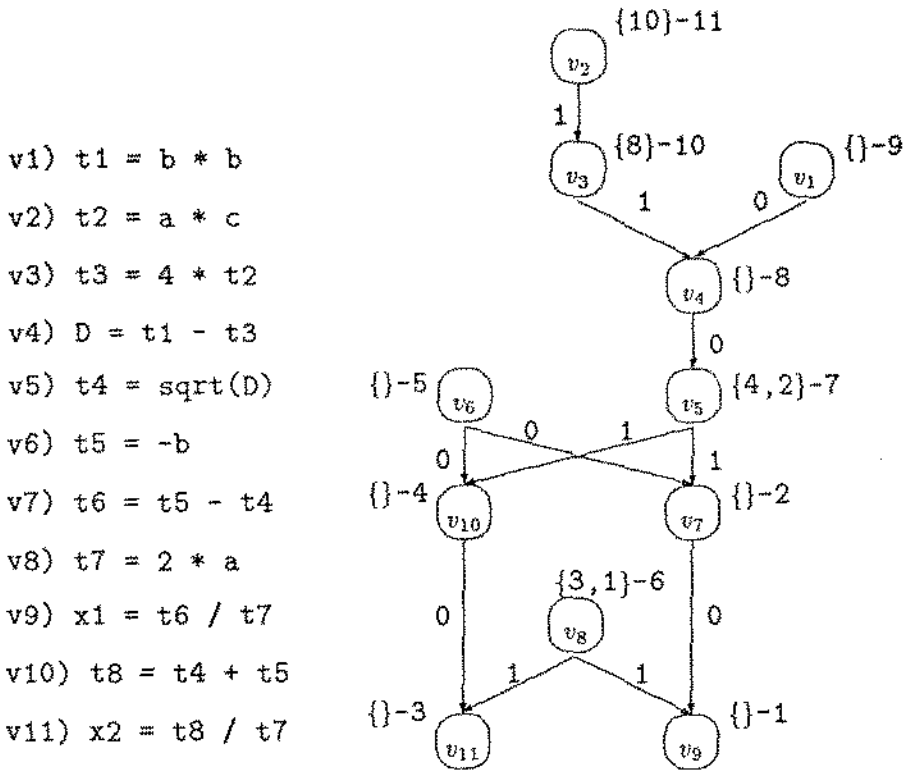


Figura 6.3: Um exemplo da aplicação do algoritmo de Bernstein.

Comentários

No artigo deles, Bernstein e Gertner não revelam qualquer experiência com a implementação do algoritmo, mas se dependências de recursos são ignoradas, as condições que eles impõem são respeitadas por muitas máquinas RISC tradicionais, inclusive código inteiro da SPARC. Ainda considerando que dependências estruturais nestas máquinas são resolvidas geralmente pelo *hardware*, um bom experimento seria verificar a efetividade deste método, enquanto heurística para o caso geral.

6.3.2 Reorganização com Dependências Estruturais de Um Ciclo

Gibbons e Muchnick [GM86] desenvolveram uma técnica heurística muito simples que é capaz de reorganizar código em máquinas com *interlocks* de um ciclo, mesmo que elas não estejam relacionadas (via arestas) no DAG de código. A técnica supõe que instruções executam em um ciclo e *interlocks* também não excedem um ciclo.

O Algoritmo Básico

O algoritmo escalona, em cada instante, um vértice do DAG de código, removendo-o a seguir. Em cada momento, as raízes (atuais) do DAG são os candidatos a escalonamento. Escolher um vértice pronto é um critério insuficiente, dadas as dependências de recursos. Resultam disto, duas regras para escolher o próximo candidato a escalonamento:

- escolher uma instrução, entre as raízes do DAG, que não tenha dependência de recursos ou dados com a última escalonada; se nenhuma existe, emitir um nop (se for o caso);
- se mais de uma existe, faça a escolha heurística priorizando as que têm dependências com os sucessores.

As heurísticas complementares que determinam o vértice escolhido para escalonamento se baseiam em quatro valores, calculados durante a construção do DAG:

- Número de sucessores que geram *interlocks* (de dados);
- Número de sucessores imediatos;
- Altura do vértice no DAG;
- Ordem no código original.

Os quatro valores são usados para critério de desempate, naquela ordem, entre um conjunto de raízes prontas. Eles refletem os seguintes raciocínios:

- Se há dependências com os sucessores, é razoável escalonar a instrução o mais cedo possível para que vértices "paralelos" sejam mais facilmente encontrados para evitar o *interlock*;
- Descobrir o maior número de sucessores gera, potencialmente, um maior número de futuras escolhas;
- Balancear o escalonamento entre vários ramos do DAG; com isto, seqüências longas que não podem se movimentar são desestimuladas;

Considere para escalonamento o DAG da Figura 6.4, onde cada um dos vértices da Figura 6.1 tem associado a si uma seqüência de quatro valores usados como critérios de desempate heurístico por Gibbons e Muchnick. Inicialmente tem-se $\{v_1, v_2, v_6, v_8\}$ como candidatos prontos. Pela primeira regra de desempate, v_8 é escolhido. Pela mesma razão v_2 vem a seguir. O escalonamento completo é: $v_8, v_2, v_3, v_6, v_1, v_4, v_5, v_7, v_{10}, v_9, v_{11}$. Os três *interlocks* da ordenação inicial foram reduzidos a um: entre v_5 e v_7.

Se além de dependências de dados, for considerado que há dependência de recursos se quaisquer duas instruções `mul` são escalonadas em seqüência, então o escalonamento resultante é: $v_8, v_6, v_2, v_3, v_1, v_4, v_5, v_7, v_{10}, v_9, v_{11}$. Ainda há três, $\{(v_2, v_3), (v_3, v_1), (v_5, v_7)\}$, *interlocks*. A ordenação original tinha quatro: $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_8, v_9)\}$. Coincidentemente, o escalonamento final de Bernstein (que não trata dependência de recursos) também tem três *interlocks*, $\{(v_8, v_3), (v_3, v_1), (v_5, v_7)\}$, se esta condição for aplicada. Analisando todos os escalonamentos permitidos pelas dependências é possível determinar que o número mínimo é um.

```
v1) t1 = b * b
v2) t2 = a * c
v3) t3 = 4 * t2
v4) D = t1 - t3
v5) t4 = sqrt(D)
v6) t5 = -b
v7) t6 = t5 - t4
v8) t7 = 2 * a
v9) x1 = t6 / t7
v10) t8 = t4 + t5
v11) x2 = t8 / t7
```

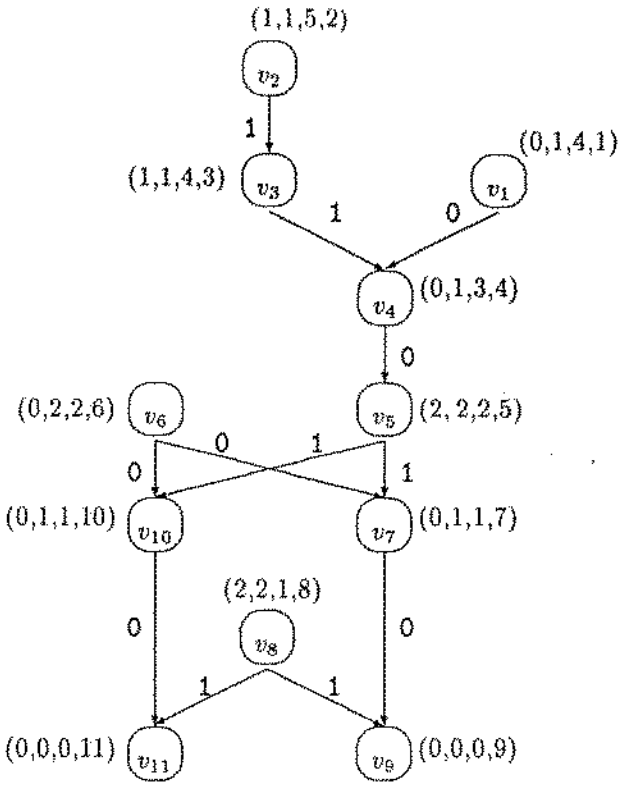


Figura 6.4: Um exemplo da aplicação do algoritmo de Gibbons e Muchnick.

Comentários

Os autores apresentam poucos números sobre a eficácia da técnica. Uma observação importante deles foi que a serialização de quaisquer instruções referenciando operandos em memória reduz significativamente as oportunidades de reorganização, sendo alguma espécie de informação de sinonímia relevante.

6.3.3 Reorganização com *Interlocks* Arbitrários

A idéia de implementar um reorganizador de instruções para a arquitetura SPARC [Sou91] mostrou que embora os algoritmos existentes oferecessem estratégias, em alguns casos de comprovado sucesso, para uma boa implementação, elas eram insuficientes em vários aspectos, mesmo sendo a SPARC uma arquitetura muito simples. Algumas das características da arquitetura não suportadas pelos escalonadores (para RISC) atuais são:

- SPARC tem instruções de múltiplos ciclos; por exemplo, instruções de ponto flutuante gastam dois ciclos na UI e mais alguns ciclos na UPF;

- determinadas instruções têm dependências estruturais de vários ciclos; um STORE, por exemplo, deve ser separado de outro por três ciclos;
- nem todas as dependências estruturais são resolvidas pelo *hardware*, sendo nops necessários;

Estas características, em conjunto com outras oriundas de máquinas recentes, motivaram o desenvolvimento de um método de reorganização capaz de escalonar instruções com custo de execução de vários ciclos, latências quaisquer, dependências estruturais e *slots* de desvios de tamanhos arbitrários. A inserção de nops também é suportada.

A técnica é uma generalização da proposta de Warren [War90]. As diferenças desta técnica em relação à de Warren são o tratamento de dependências de recursos e a possibilidade de inserção de nops. Em função das características da arquitetura para a qual projetou originalmente o seu método, Warren também ignorou *slots* de desvios.

O Algoritmo Básico

Dado o conjunto de recursos, R, da arquitetura, uma instrução, i, é reconhecida através dos seguintes atributos:

- o custo da instrução, $CUSTO(i)$ (vide definição no capítulo 2, página 15);
- a latência da instrução, $LAT(i)$; um valor negativo indica que não há necessidade de preencher explicitamente ciclos de retardo com instruções nop;
- o número de *slots* da instrução, se ela muda o fluxo normal da execução.
- um valor, $U_i(r), \forall r \in R$. Se o recurso r é utilizado pela instrução, então $U_i(r) = 1$; zero, caso contrário;
- um número inteiro, $BR_i(r), \forall r \in R$, representando o número de ciclos que devem separar esta instrução de uma instrução subseqüente que utiliza r. Um número negativo tem a mesma conotação anterior.

O algoritmo baseia-se no cálculo do menor ciclo a partir do qual cada instrução v no grafo de dependências G pode ser escalonada ($t\_esc(v)$), sem gerar atrasos. No início do escalonamento $t\_esc(v) = 0, \forall v \in G$. Para controlar o número de ciclos após um certo número de escalonamentos, existe a variável *t_corrente*, inicialmente $t\_corrente = 0$.

Em algum instante do escalonamento podem existir várias ou nenhuma instrução pronta. O primeiro caso é resolvido pelas seguintes heurísticas (bem estabelecidas na literatura [GM86, War90]):

- priorizar instruções com maior distância das folhas no grafo de dependências; o cálculo da distância poderia considerar a latência das instruções como pesos das arestas;
- dar oportunidade a instruções com o maior número de sucessores;
- escolher a que vem primeiro na ordem original.

Quando nenhuma instrução está pronta, a que se tornará pronta mais brevemente, v, é selecionada. Observe-se, entretanto, que isto implica em incrementar *t_corrente* até que ele atinja o tempo mínimo exigido para escalonamento de v, $t\_esc(v)$. Este incremento pode, por sua vez, exigir a inserção de nops, pois alguma espécie de separação explícita pode ser requerida entre uma instrução escalonada anteriormente e a que acaba de ser selecionada. Para resolver isto, cada instrução i tem associado um valor $t\_nop(i)$ indicando o menor ciclo onde ela pode ser escalonada sem necessidade de inserção de nop; em qualquer instante vale a desigualdade: $t\_esc(i) \geq t\_nop(i), \forall i \in G$. O número de nops emitidos nestes casos é a diferença entre o *t_nop* da instrução escolhida e o *t_corrente* (antes de ser incrementado).

Valores de *t_esc* e *t_nop* também estão associados a cada recurso, a fim de controlar as dependências associadas a eles. Tais valores são necessários porque instruções que acabam de se tornar "prontas" (se apenas dependências de dados são consideradas) podem ter dependências de recursos com instruções anteriormente escalonadas. Isto exige que, após cada escalonamento, os valores de *t_esc* e *t_nop* de cada instrução virtualmente pronta sejam atualizados em função dos recursos bloqueados, desde que os utilize, obviamente.

Escalonada uma instrução v_i, o *t_corrente* é atualizado:

$$t\_corrente = t\_corrente + CUSTO(v_i)$$

O escalonamento de instrução v_i tem também o efeito de aumentar o valor de *t_esc* de todos os seus sucessores (por dependência de dados), v_s:

$$t\_esc(v_s) = MAX(t\_esc(v_j),\ t\_corrente + |LAT(v_i)|)$$

$$t\_nop(v_s) = MAX(t\_nop(v_j),\ t\_corrente + LAT(v_i))$$

O número de ciclos que v_i bloqueia recursos serve para definir os novos valores de *t_esc* e *t_nop* para cada recurso $r \in R$.

$$t\_esc(r) = MAX(t\_esc(r),\ t\_corrente + |BR_{v_i}(r)|)$$

$$t\_nop(r) = MAX(t\_nop(r),\ t\_corrente + BR_{v_i}(r))$$

Finalmente para todas as instruções v que já tiveram todos os seus predecessores escalonados, as dependências de recursos devem ser evitadas alterando convenientemente os valores de *t_esc* e *t_nop*:

$$t\_esc(v) = MAX(t\_esc(v),\ t\_esc(r)) \quad \forall r \in R,\ U_v(r) = 1$$

$$t\_nop(v) = MAX(t\_nop(v),\ t\_nop(r)) \quad \forall r \in R,\ U_v(r) = 1$$

Estes novos valores, calculados após cada escalonamento, são então usados para guiar a seleção do próximo candidato a escalonar.

Considere novamente o escalonamento do grafo da Figura 6.1. Dado que não tem sido considerado o uso explícito de nops seu tratamento será omitido na discussão que segue, o mesmo acontece para dependências de recursos. Inicialmente $t\_esc(v), \forall v \in G$, e *t_corrente* valem zero. O primeiro candidato escolhido entre o conjunto de prontos $\{v_1, v_2, v_6, v_8\}$ é v_2 por estar mais distante das folhas. Então, v_2 libera v_3, mas inicia $t\_esc(v_3) = 2$, impedindo-lhe de ser escalonado a seguir,

dado que *t_corrente* vale um. Pela mesma heurística v_1 é escalonado a seguir. O escalonamento completo é: $v_2, v_1, v_3, v_6, v_4, v_5, v_8, v_7, v_9, v_{10}, v_{11}$, que não possui *interlocks*.

Sob a condição de que duas instruções `mul` não podem ser executadas em seqüência, o algoritmo também gera um escalonamento ótimo (contendo um único *interlock*). Para modelar a dependência estrutural cria-se um recurso, $RMUL$, que é usado e bloqueado (durante um ciclo) após a execução da instrução `mul`. Como antes v_2 é inicialmente escalonado, mas desde que $t\_esc(RMUL) = 2$, $t\_esc(v_1) = t\_esc(v_3) = t\_esc(v_8) = 2$, restando escolher v_6 ($t\_esc(v_6) = 1$). A seguir v_3 é escolhido pelas heurísticas. Nesse ponto, $t\_esc(RMUL) = 4$ e nenhum vértice se torna pronto, determinando o único *interlock* do escalonamento final: $v_2, v_6, v_3, v_1, v_4, v_5, v_8, v_7, v_9, v_{10}, v_{11}$.

Tratamento de Desvios

Blocos básicos comumente terminam com uma instrução de desvio e arquiteturas RISC normalmente possuem *delayed-branch*; por estas razões se resolveu incluir no algoritmo de escalonamento o tratamento deste mecanismo. Em geral o número de *slots* não excede uma instrução, ainda assim o caso geral é suportado.

A diferença de escalonar uma instrução de desvio é que mesmo estando pronta seu escalonamento deve garantir que o número de instruções que a sucedem seja exatamente o número de *slots*. Uma técnica simples e que funciona bem para desvios incondicionais consiste em não considerar o desvio durante o escalonamento e depois, supondo k *slots*, inseri-lo entre a *k-ésima* e a $(k+1)$*-ésima* instruções, contadas do final para o início do escalonamento resultante. Para desvios condicionais e no contexto que o algoritmo acima trabalha, esta técnica ainda pode ser aplicada, contudo alguns cuidados devem ser tomados.

A idéia é continuar o processo de reorganização como anteriormente, mas nunca escolher o desvio. Assim o desvio sai do escalonamento com o *t_esc* a partir do qual poderia ser escalonado. Adicionalmente, todas as demais instruções devem manter em seus *t_esc* o ciclo onde foram escalonadas.

Em um segundo passo a posição onde o desvio pode ser inserido é localizada percorrendo o escalonamento da última à primeira instrução enquanto as seguintes condições são respeitadas:

- o *t_esc* do desvio é menor do que ou igual ao *t_esc* da instrução correntemente sob verificação;
- o número de instruções já verificadas é menor do que número de *slots*;

A instrução de desvio é inserida imediatamente depois da última instrução analisada. Algumas correções podem, contudo, ser úteis ou necessárias. Por exemplo, se apenas a segunda condição falhou e a instrução que precede a posição onde o desvio foi inserido é um nop, este pode ser removido, pois o desvio tomará a posição de separador que o nop representa. Mas se apenas primeira falhou então um certo número de nops deve ser inserido para que o número de instruções seguindo o desvio possa ser atingido.

A altura do desvio é iniciada com o valor correspondente ao seu número de *slots*, priorizando, ligeiramente, o escalonamento dos seus predecessores e garantindo um maior número de candidatos para preencher os *slots*.

Comentários

O próximo capítulo descreve os resultados obtidos com uma implementação deste algoritmo para a SPARC. Apesar da generalidade, ele não trata dependências de recursos que acontecem em ciclos específicos após o início da execução de uma instrução; por exemplo, o controle de acesso ao barramento no Motorola 88000 [Mel89].

Como um trabalho futuro, pretendemos verificar a aplicabilidade da técnica para arquiteturas VLIW e superescalares. As mudanças parecem simples quando não existem múltiplas unidades de um mesmo tipo. Sob esta condição, o escalonamento em superescalares pode ser modelado fazendo cada instrução ter a unidade onde executa em seu conjunto de recursos usados e bloqueá-la por tantos ciclos quantas são as outras unidades, forçando assim o escalonamento de instruções executáveis nelas.

6.4 Reorganização além de Blocos Básicos

Vários estudos, também aqui confirmados (vide capítulo 7), indicam que o número de instruções por bloco básico é, em geral, pequeno. O grau de paralelismo das instruções é ainda menor.[4] Portanto, máquinas com várias unidades funcionais, capazes de executar instruções em paralelo, podem ser pouco efetivas se o escalonamento restringe-se a código linear.

Alguns experimentos, entretanto, mostram que se o caminho tomado pelo programa (em desvios condicionais) fosse conhecido *a priori*, o grau de paralelismo seria consideravelmente maior [Wal91] e, portanto, arquiteturas explorando paralelismo de baixa granularidade poderiam ter grande *speed up*. Nesta seção apresentam-se três técnicas que aumentam, implícita ou explicitamente, o tamanho dos blocos básicos, objetivando aumentar o paralelismo e conseqüentemente o desempenho de máquinas como VLIW e superescalares. A primeira, desmembramento de malhas, não foi projetada originalmente para aumentar paralelismo, porém também se presta a esta finalidade em alguns casos. *Trace scheduling* é uma técnica bem estabelecida e usada em compiladores de produção para VLIW. Uma introdução ao conceito de *pipeline* de *software* encerra esta seção.

6.4.1 Desmembramento de Malhas

Desmembramento de malhas é uma otimização clássica envolvendo um compromisso entre tamanho e tempo de execução do código emitido para uma estrutura repetitiva. Aplicar o desmembramento consiste em replicar o corpo da malha, alterando convenientemente os índices que controlam a execução da malha. A Figura 6.5 mostra o efeito da otimização sobre uma malha escrita em C.

Comentários

Enquanto otimização usual, desmembramento é muito restrito, pois só é útil quando o tempo gasto executando o controle da malha é significativo em relação ao tempo gasto executando o corpo da malha. Some-se a isto que em máquinas com pequeno *cache* de instruções, o tempo de execução pode até mesmo ser aumentado se esta "otimização" é aplicada.

[4] Grau de paralelismo nesse sentido indica o número médio de instruções executáveis, em cada ciclo, numa arquitetura com infinitos recursos.

```
for (i=0; i < 15; i++)
   v[i] = 0;
```

(a)

```
for (i=0; i < 15; i+=3) {
   v[i] = 0;
   v[i+1] = 0;
   v[i+2] = 0;
}
```

(b)

Figura 6.5: Uma malha escrita em C: (a) antes e (b) após aplicado o desmembramento.

Objetivando explorar paralelismo de baixa granularidade, replicar o código não corresponde por si só a um aumento do grau de paralelismo, visto que o corpo da malha pode ter mais de um bloco básico. De certa forma, desmembramento serve como base para aumentar a efetividade de técnicas que exploram o paralelismo entre instruções de blocos básicos diferentes.

6.4.2 Trace Scheduling

Trace scheduling [Fis81] procura aumentar o grau de paralelismo considerando para reorganização os caminhos (*traces*) mais executados em seqüências de blocos básicos que não contém ciclos. Dentro de um *trace*, instruções podem se movimentar sem restrições quanto ao fluxo de controle, ou seja, a reorganização pode ser executada por um algoritmo de escalonamento para blocos básicos. Posteriormente, pode ser necessário emitir instruções de compensação que corrigem alguns efeitos deste processo, dada a possibilidade do *trace* não vir a ser executado na sua totalidade. Observe-se que as instruções de desvio continuam a fazer parte do *trace*, apenas não funcionam como limites para a reorganização.

As condições e regras que controlam a movimentação de instruções entre blocos básicos são apresentadas na Tabela 6.1 e referem-se ao grafo de fluxo de controle da Figura 6.6. Mesmo sendo o grafo da Figura 6.6 muito simples, ele também é genérico o suficiente para capturar todos os tipos de movimentação na ausência de ciclos e por esta razão será usado na descrição de *trace scheduling*. Sempre que algum aspecto for relevante à generalização da técnica para grafos arbitrários, será mencionado.

Note-se que *trace scheduling* não trabalha aplicando explicitamente as regras de movimentação entre blocos básicos, mas sim considerando trechos de reorganização que incluem desvios e corrigindo distorções causadas pelas movimentações que cruzam desvios. Existe, porém, uma seqüência de movimentações que conduzem aos mesmos resultados de *trace scheduling*.

O Algoritmo Básico

Aplicar *trace scheduling* equivale a, repetidamente, executar três atividades: selecionar um *trace*, aplicar um algoritmo de reorganização e emitir código de compensação de modo a respeitar os critérios da Tabela 6.1. A repetição é necessária porque os trechos de programa não selecionados no *trace* podem conter seqüências de blocos básicos e, portanto, seus próprios *traces*. A descrição de cada uma das fases é feita a seguir:

Regra	MOVIMENTAÇÃO		Condições
	de	para	
1	$B2$	$B1$ e $B4$	A instrução é raiz do DAG de código de $B2$
2	$B1$ e $B4$	$B2$	Cópias idênticas da mesma instrução são folhas dos DAGs de código de $B1$ e $B4$
3	$B2$	$B3$ e $B5$	A instrução é folha do DAG de código de $B2$
4	$B3$ e $B5$	$B2$	Cópias idênticas da mesma instrução são raízes dos DAGs de código de $B3$ e $B5$
5	$B2$	$B3$ (ou $B5$)	A instrução é folha do DAG de código de $B2$ e os registradores (memória) por ela definidos não estão vivos em $B5$ ($B3$).
6	$B3$ (ou $B5$)	$B2$	A instrução é raiz do DAG de código de $B3$ ($B5$) e os registradores (memória) por ela definidos não estão vivos em $B5$ ($B3$).

Tabela 6.1: Critérios para movimentação de instruções entre blocos básicos da Figura 6.6.

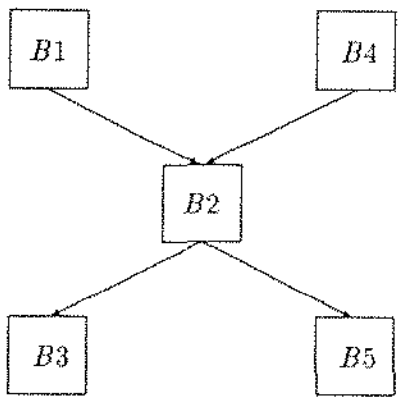


Figura 6.6: Grafo de fluxo de controle "genérico".

Seleção de traces Para determinar o trecho de código mais freqüentemente executado é necessário conhecer a probabilidade de execução de cada um dos ramos das instruções de desvio. Isto pode ser obtido através de *profile* ou de informação fornecida pelo programador. A partir do bloco básico de maior freqüência de execução estimada, o *trace* cresce em ambos os sentidos.

Para trás, o crescimento se dá escolhendo entre os predecessores do bloco básico que inicia o *trace* até então formado (b_i), aquele que mais contribui para a freqüência de execução de b_i. O bloco escolhido não deve pertencer a *traces* anteriormente formados, de outra forma, instruções de blocos básicos pouco executados (*trace* sendo formado) poderiam vir a fazer parte de blocos básicos mais freqüentemente executados (*traces* já formados). Se esta condição não pode ser respeitada, o processo pára.

O crescimento para frente consiste em escolher entre os sucessores do último bloco até então no *trace*, aquele para onde, mais comumente, este desvia o fluxo de controle. A condição de parada é semelhante.

Para fins de exemplo, no restante desta subseção considera-se que o *trace* selecionado na Figura 6.6 é composto de $\{B1, B2, B3\}$.

Reorganização Selecionado o *trace*, um grafo de dependências para todo o *trace* é construído. Algumas arestas adicionais devem, entretanto, ser adicionadas ao DAG. Em particular, a regra 6 da Tabela 6.1 impõe a condição de que instruções que redefinem registradores (memória) em $B3$ não podem ser movimentadas para $B2$ quando estes estão vivos em $B5$. A maneira natural de respeitar esta condição é inserir arestas partindo da instrução de desvio no fim de $B2$ para aquelas em $B3$ que (re)definem recursos vivos em $B5$. Arestas também são inseridas para evitar a movimentação inadequada de instruções que podem causar exceções de *hardware*, por exemplo, divisão por zero ou raiz quadrada de números negativos (pode não ser fácil detetar tais casos). Neste ponto, o *trace* passa a ser visto como um único bloco básico, podendo um algoritmo de reorganização compatível ser empregado.

Código de compensação A fim de manter a equivalência semântica do trecho de programa submetido a reorganização, identifica-se na Figura 6.6 dois tipos de pontos de programa: pontos de junção – referem-se a pontos do *trace* que são alvos de desvios partindo de fora do *trace*; pontos de divisão – correspondem aos pontos que seguem cada um dos desvios condicionais no *trace*.

Em relação aos pontos de junção, se uma instrução inicialmente em $B1$ foi movida para $B2$, o alvo do desvio condicional em $B4$ deve ser modificado. Isto é necessário porque não se deve executar qualquer instrução de $B1$ quando o fluxo de execução passa por $B4$. Ao deslocar o alvo do desvio, algumas instruções de $B2$ (ou mesmo de $B3$) podem deixar de ser executadas, também em função da reorganização sobre o *trace*, criando a necessidade de introduzir código de compensação. Sumarizando estes requisitos, a semântica permanece inalterada se as seguintes ações são executadas (após a reorganização):

- criação de um novo bloco básico ($B2'$);
- fazer a última instrução de $B2'$ ser um desvio para o primeiro ponto após o escalonamento de todas as instruções de $B1$.
- copiar para $B2'$ todas as instruções de $B2$ e $B3$ escalonadas acima do novo alvo, na mesma ordem do escalonamento;
- fazer o alvo de $B4$ ser o ponto inicial de $B2'$.

A Figura 6.7 mostra a posição lógica de $B2'$ no referido grafo de fluxo de controle. Observe que os blocos básicos identificados na Figura não necessariamente contêm as mesmas instruções que antes da reorganização.

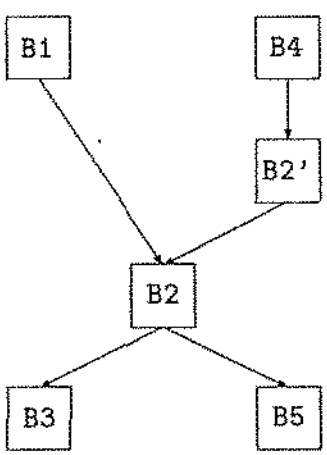


Figura 6.7: Efeitos da emissão de código de compensação em *trace scheduling* - parte 1.

No caso de pontos de divisão, por exemplo o ponto final de $B2$, é possível que instruções originalmente acima deste, estejam abaixo após a reorganização, exigindo também a emissão de código de compensação. Note que se o contrário acontece, a emissão não é necessária,

pois as arestas introduzidas na fase de reorganização garantem as condições da regra seis. As ações para manter a semântica original são:

- criação de um novo bloco básico ($B5'$);
- fazer a última instrução de $B2$ desviar condicionalmente para o ponto inicial de $B5'$
- copiar para $B5'$ todas as instruções de $B1$ e $B2$ escalonadas abaixo do ponto final de $B2$, na mesma ordem do escalonamento;
- fazer o alvo de $B5'$ ser o ponto inicial de $B5$.

A Figura 6.8 mostra a posição lógica de $B5'$ no referido grafo de fluxo de controle. Observe que os blocos básicos identificados na Figura não necessariamente contêm as mesmas instruções que antes da reorganização.

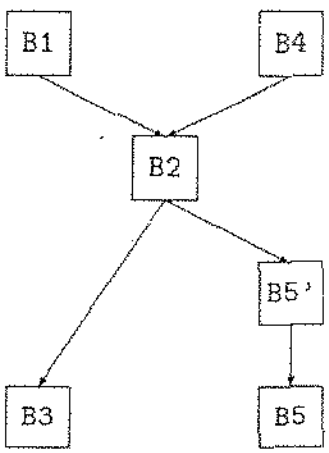


Figura 6.8: Efeitos da emissão de código de compensação em *trace scheduling* - parte 2.

Trace scheduling e as Regras de Movimentação

Os três passos de *trace scheduling* executam sistemática e transparentemente as movimentações de código expressas nas regras um, três, cinco e seis.

No contexto de *trace scheduling*, aplicar a regra dois consiste em permitir que um ponto de junção não suceda todas as instruções que anteriormente o precediam. Em outros termos, na Figura 6.6, o alvo do desvio no final de $B4$ pode estar antes de uma instrução originalmente de $B1$. Para isso, ela deve ser a última instrução escalonada de $B1$, estar replicada e ser folha do DAG de código de $B4$.

Similarmente, se existem instruções idênticas em $B3$ e $B5$ e ambas são raízes de seus respectivos DAGs de código, é suficiente, para empregar a regra quatro, não inserir arestas extras que forçam o escalonamento desta instrução após o desvio no final de $B2$, como recomendado anteriormente. Se após a reorganização a instrução em $B3$ precede o desvio em $B2$, então a cópia presente em $B5$ pode ser removida. Se um outro bloco B atingisse $B5$, então um novo bloco precisaria ser colocado entre B e $B5$, contendo apenas a instrução removida.

Comentários

Por ser restrito a grafos de controle livres de ciclos, *trace scheduling* não descobre paralelismo entre iterações distintas de uma mesma malha. Em grafos de controle de fluxo redutíveis,[5] entretanto, *trace scheduling* pode ser aplicado da malha mais interna até a malha mais externa. Em malhas que contêm outras, estas são substituídas por uma pseudo-instrução que as representam (todas as dependências são especificadas), permitindo que algum paralelismo entre instruções intercaladas por malhas possa ser encontrado. Este processo é as vezes chamado de redução hierárquica.

Empregar *trace scheduling* para aumentar o paralelismo de baixa granularidade, exige algum suporte de *hardware* para tratamento de desvios. Como já citado, blocos básicos são muito pequenos e podem terminar com alguma espécie de desvio. Se nenhum técnica de execução de desvios for considerada, após a aplicação de *trace scheduling*, os desvios permanecem ocupando instruções longas seqüencialmente e degradam as vantagens obtidas com a movimentação das instruções. Um mecanismo elegante foi projetado para a máquina VLIW Multiflow [CNO+88]: despacho de múltiplas instruções de desvio simultaneamente, com prioridades relativas entre eles. Com o empacotamento de muitas instruções de desvio em uma única instrução longa é necessário evitar que instruções que seguem logicamente o primeiro desvio tomado (entre os empacotados) armazenem os seus resultados. Isto é uma conseqüência da regra seis e pode ser visto como uma espécie de generalização do princípio de anulação, implementado na SPARC (capítulo 3).

Vários outros problemas surgem com *trace scheduling*. Primeiro, pode ser difícil estimar a probabilidade de um desvio ser tomado ou não; segundo, o tratamento insatisfatório de malhas pode demandar o uso de desmembramento, com os problemas que ele acarreta. O terceiro e mais importante é a explosão de código. Gasperoni [Gas89] menciona que o código de compensação pode ser exponencial no número de instruções do procedimento de entrada; na prática este número parece justificar o aumento de desempenho obtido. Alguns experimentos sobre "código científico" executando em uma máquina com oito UI e oito UPF, mostraram ([Gas89]) um aumento de desempenho por um fator 5, enquanto o tamanho do código cresceu por um fator de seis.

6.4.3 *Pipeline* de *Software*

Malhas são, para algumas arquiteturas, a fonte primária de obtenção de paralelismo. O objetivo é, de alguma forma, iniciar a execução de uma iteração antes do término daquelas que a precedem. Em máquinas assíncronas [HB84], isto equivale a atribuir uma iteração a cada processador. Nas máquinas que se tem analisado neste trabalho, VLIW e superescalares, explorar o paralelismo em malhas consiste em ter instruções de várias iterações executando simultaneamente em um mesmo processador (com várias unidades funcionais).

A partir de uma ordem de avaliação para o corpo de uma malha, várias iterações podem ser iniciadas em intervalos regulares criando a idéia de que cada iteração está em uma fase (estágio) de processamento, um conceito semelhante a um *pipeline* de instruções.

Considere a malha mostrada na Figura 6.9(a), com representação intermediária na parte (b) da mesma Figura. Quando a simulação da execução paralela é efetuada, chega-se à Figura 6.9(c), onde uma iteração é iniciada um ciclo após a anterior. Observa-se que a partir do quinto ciclo um padrão se repete, com uma instrução de cada iteração sendo executada ao mesmo tempo. Intuitivamente isto corresponde a um *pipeline* de *software*. As instruções da malha que precedem e sucedem a

[5] Programas estruturados conduzem a grafos de controle redutíveis.

parte repetitiva, ou estado operacional, do *pipeline* denominam-se prólogo e epílogo do *pipeline*, respectivamente.

```
i = 0;
do {
   i++;
   s = s + v[i];
} while (i < 100);
```

(a)

```
   mov  0, r_i
L: add  r_i, 1, r_i
   ld   "v[r_i]", r1
   add  r_s, r1, r_s
   cmp  r_i, 100
   b_< L
E:
```

(b)

CICLO	ITERAÇÃO						
	1	2	3	4	5	6	7
0	mov 0,r_i						
1	add r_i,1,r_i						
2	ld "v[r_i]",r1	add r_i,1,r_i					
3	add r_s,r1,r_s	ld "v[r_i]",r1	add r_i,1,r_i				
4	cmp r_i,100	add r_s,r1,r_s	ld "v[r_i]",r1	add r_i,1,r_i			
5	b_< L	cmp r_i,100	add r_s,r1,r_s	ld "v[r_i]",r1	add r_i,1,r_i		
6		b_< L	cmp r_i,100	add r_s,r1,r_s	ld "v[r_i]",r1	add r_i,1,r_i	
7			b_< L	cmp r_i,100	add r_s,r1,r_s	ld "v[r_i]",r1	add r_i,1,r_i
8				b_< L	cmp r_i,100	add r_s,r1,r_s	ld "v[r_i]",r1
9					b_< L	cmp r_i,100	add r_s,r1,r_s
10						b_< L	cmp r_i, 100
11							b_< L

(c)

Figura 6.9: Um exemplo de *pipeline* de *software*.

A principal tese envolvida com *pipeline* de *software* é que iterações podem ser iniciadas em intervalos constantes e, tentativamente, sem esperar o término das iterações precedentes. O problema consiste, então, em determinar este intervalo de iniciação t_i entre as iterações. É possível concluir que quanto menor o intervalo de iniciação mais eficiente é o *pipeline* de *software*, pois o intervalo de iniciação é igual ao número de ciclos gastos no estado operacional do *pipeline*. No pior caso, o valor de t_i é igual ao número de instruções, indicando que o *pipeline* de *software* não pode ser conseguido.

Note-se também que o estado operacional "empacota" exatamente o mesmo número de instruções de uma iteração. Como o cálculo do número mínimo de instruções nop que elimina dependências de dados entre instruções de uma mesma iteração é um problema *NP-completo*, encontrar um *pipeline* de *software* ótimo também o é. Aiken e Nicolau [AN88], no entanto, provaram que na presença de ilimitados recursos, há um algoritmo polinomial que encontra o *pipeline* de *software* ótimo; a existência do algoritmo se deve ao fato de que com ilimitados recursos, os escalonamentos podem ser distintos para cada iteração, eliminando a causa da intratabilidade do problema.

Nesta seção apresenta-se um algoritmo para geração de um *pipeline* de *software* a partir do escalonamento de uma iteração.

O Algoritmo Básico

Um algoritmo simples, devido a Charlesworth [Cha82b], generaliza a idéia desenvolvida na Figura 6.9. Exatamente, o algoritmo:

1. encontra um escalonamento para uma iteração; após o escalonamento as n instruções são numeradas de i_0 até i_{n-1}. O intervalo de iniciação t_i é supostamente 1;
2. tenta empacotar as n instruções da malha em t_i instruções "longas". A *i-ésima* instrução deve estar na instrução longa $i \ mod \ t_i$. Se dependências de recursos impedem o empacotamento, o valor de t_i é somado de um e este passo repetido;
3. gera o prólogo e o epílogo do *pipeline* de *software*.

Após descoberto o intervalo de iniciação, alguns ajustes em índices de vetores ou variáveis de controle da malha podem ser necessários devido a atualizações de variáveis executadas por instruções de outras iterações. Na Figura 6.9, por exemplo, quatro iterações são iniciadas antes que o estado operacional seja atingido, restando iniciar 96; desde que o valor de `r_i` no instante da primeira avaliação da condição é três, a comparação deve ser mudada para 99.

A limitação do algoritmo de Charlesworth é não tratar sistematicamente complicações advindas de dependências de dados entre instruções de iterações diferentes. Por exemplo, em:

```
for (i=1; i < 100; i++)
   s[i] = s[i - 1] + a[i];
```

Considerar dependências inter-iterações cria complicações. A principal é que o grafo de dependências pode ser tornar cíclico. A exposição de algoritmos que tratam o caso geral está além do escopo deste trabalho; Lam [Lam88] e Gasperoni [Gas89] o fazem. Os problemas relacionados a circularidades no grafo de dependências são comuns em compiladores vetorizadores, mas ao contrário do problema de vetorização, uma circularidade não implica que a malha não pode ser *pipelined*, pois as instruções não são executadas de forma indivisível como em operações vetoriais.

Comentários

Idealmente o aumento de desempenho de um *pipeline* de *software* é $\lceil \frac{n}{t_i} \rceil$, que é o número de instruções executadas em paralelo. Este aumento pode ser pequeno quando existem muitas dependências entre as iterações, pois t_i tende a se aproximar de n. Por outro lado, o próprio empacotamento das n instruções de uma iteração em instruções "longas", já dá algum aumento de desempenho.

Lam fez vários experimentos e constatou que *pipeline* de *software* dá um aumento médio de desempenho da ordem de três sobre código empacotado apenas em blocos básicos; o tamanho do código também aumentou em três vezes, na média. Ela usou redução hierárquica para movimentar instruções dentro de uma mesma iteração.

6.5 Integração entre Alocação de Registradores e Reorganização de Instruções

Na seção anterior foram abordados alguns métodos de aumentar o paralelismo de baixa granularidade como uma forma de incrementar a eficiência de máquinas com capacidade de alimentar mais

de uma unidade funcional simultaneamente. Basicamente o paralelismo é aumentado tornando maiores as seqüências de código onde um reorganizador de instruções pode ser aplicado.

Como já abordado, observa-se que alocação de registradores pode reduzir as oportunidades de reorganização (conseqüentemente o paralelismo) se aplicado antes do reorganizador e, vice-versa; compiladores atuais têm dado pouca ênfase a estes relacionamentos. Um exemplo pode ser visto na Figura 6.10, onde a parte (a) mostra o código SPARC gerado pelo GCC para somar três variáveis inteiras (todas em memória) e atribuir o resultado a uma delas. A parte (b) mostra um código alternativo, que usa um registrador a mais que o necessário, porém diminui o tempo de execução em dois ciclos. Visto de outra forma, o grau de paralelismo da primeira seqüência é 1 e da segunda 1,5 (vide nota de rodapé na página 90).

```
ld    ["a"], r1
ld    ["b"], r2
add   r1, r2, r2
ld    ["c"], r1
add   r1, r2, r2
st    r2, ["b"]
```

(a)

```
ld    ["a"], r1
ld    ["b"], r2
ld    ["c"], r3
add   r1, r2, r2
add   r2, r3, r2
st    r2, ["b"]
```

(b)

Figura 6.10: Código SPARC para somar três variáveis: a) usa dois registradores e gasta 13 ciclos; b) usa três registradores e gasta 11 ciclos.

Mesmo quando a exploração de algum paralelismo oferecido pelo *hardware* não é o objetivo principal, a coordenação entre alocação e reorganização é importante porque em algumas arquiteturas instruções de ponto flutuante têm latência de vários ciclos, aumentando a necessidade de instruções não relacionadas no grafo de dependências para preencher os ciclos de atraso e também porque código científico usa muitas instruções de ponto flutuante.

Nesta seção dois métodos para integração das tarefas são apresentados. O primeiro é implementado em *hardware* no IBM RS/6000; o segundo foi proposto por Bradlee [Bra91] e implementado em compiladores para várias máquinas.

6.5.1 Remapeamento em *Hardware*

Um dos princípios que guiaram o projeto da arquitetura superescalar IBM RS/6000 foi que as duas unidades de processamento (UI e UPF) deveriam ser tão independentes quanto possível, embora todo o acesso à memória seja controlado pela UI e resultados de ponto flutuante possam demorar alguns ciclos para serem gerados (em função de um *pipeline* mais profundo). O princípio conduziu às seguintes características: 1) instruções de STORE não bloqueiam a UI quando o resultado de uma computação de ponto flutuante ainda não está disponível, ao invés um "processador" de STORE é ativado sempre que um dado pendente se torna pronto; 2) *buffers* de instrução permitem o despacho de várias instruções sem que a execução de algumas das precedentes tenha sido iniciada, possivelmente por causa de dependências de dados (a arquitetura resolve *interlocks*). A conseqüência dessa estrutura é que a maior latência na UPF pode atrasar consideravelmente a UI quando LOADs para registradores de pronto flutuante (RPF), ainda em uso na UPF, já tiveram todas as instruções que (fisicamente) a precedem já foram despachadas.

Para resolver o problema, um esquema de remapeamento de RPF foi implementado em *hardware*. Na visão do programador, a arquitetura tem 32 RPFs, mas internamente eles são 40. Em cada instante há uma associação entre registradores externos e internos. Quando um RPF é carregado, um novo registrador interno lhe é atribuído, se o atual está em uso na UPF. Para garantir que "usuários" de um registrador externo recebam o valor correto, a associação de externo para interno é feita antes da execução de um novo LOAD e o registrador interno permanece bloqueado até o último uso.

Comentários

Embora satisfaça o requerimento de independência entre as unidades, o mecanismo implementado não resolve o problema de reutilização de registradores por instruções aritméticas, continuando alguma espécie de integração em *software* recomendável.

Muitas destas idéias foram originalmente desenvolvidas por Tomasulo para o IBM 360/91 [Tom67]. No caso do remapeamento de RPFs, a aparente vantagem está na implementação de instruções vetoriais em *software*. Como o acesso à memória dificilmente é bloqueado e instruções podem ser despachadas a cada ciclo (não há anti-dependências), uma taxa de processamento muito alto pode ser conseguida, gerando também, em um certo sentido, um *pipeline* de *software*. No entanto, o alto desempenho do RS/6000 nos SPEC *benchmarks* de ponto flutuante, que são altamente vetorizáveis [SPE91], parecem ser mais conseqüência da baixa latência das instruções que propriamente do remapeamento.

6.5.2 Alocação com Custo de Reorganização

Recentemente, Bradlee [Bra91] desenvolveu uma técnica que integra alocação e reorganização como parte de um trabalho maior sobre derivação automática de reorganizadores de instruções para RISCs.

A técnica busca quantificar o custo de escalonar cada bloco básico em função do número de registradores associados às variáveis locais (ao bloco). Dados os lucros, é possível fazer decisões de derramamento não somente baseado no número de referências, mas no custo das referências, pois de acordo com o registrador associado a uma variável ela pode ser, ou não, atrasada por anti-dependências.

O Algoritmo Básico

O processo de alocação dos registradores é dividido em duas etapas. A alocação de variáveis cuja vida se estende por mais de um bloco básico é feita seguindo o método de Chaitin, enquanto as demais são alocadas pelo escalonador à medida que faz a reorganização. O argumento de Bradlee é que as instruções envolvendo temporários gerados pelo compilador (locais) são as que têm maior liberdade de movimentação e por isto as mais adequadas para preencher ciclos de *interlock*. A movimentação pode, contudo, aumentar as necessidades de registradores e o reorganizador é o elemento que pode precisar tais necessidades.

Para cada bloco básico, antes da alocação global, o escalonador deveria ser executado a fim de definir o custo de execução do bloco quando determinado número de registradores está disponível para alocação local. A invocação do reorganizador para todos os casos é impraticável, assim uma função é empregada para determinar o custo, dado um determinado número de registradores. O

custo é aferido em dois pontos: com um número mínimo e com a totalidade de registradores. A partir de conjecturas, a seguinte função foi estabelecida para dar a estimativa do custo de uma execução do bloco básico b, quando n_reg registradores estão disponíveis:

$$custo\_esc_b(n\_reg) = x + y/n\_reg^2$$

onde x e y são calculados a partir das aferições. Um outro resultado obtido da execução do reorganizador nesta fase é o número ideal de registradores necessários no bloco b, MR_b (forçosamente menor do que ou igual à totalidade dos registradores).

O grafo de interferências para alocação global é constituído de dois tipos de vértices: um para cada candidato não local e, para cada bloco básico b, MR_b vértices representando as necessidades do reorganizador para b. Os MR_b vértices interferem entre si e também com todos os candidatos não locais vivos em b. As demais arestas são resultado de interferências entre candidatos não locais.

A coloração segue o método de simplificação proposto por Chaitin. Entretanto, durante um bloqueio, a heurística que determina o candidato a ser derramado difere de acordo com o tipo do candidato. Se o vértice representa um candidato não local, nada muda (vide página 44). Mas para um candidato local, o custo de derramamento mede, na verdade, a penalidade imposta pelo reorganizador por dispor de um registrador a menos para fazer o escalonamento. O custo é o mesmo para todos os candidatos c, pertencentes a um mesmo bloco básico b, visto que eles não representam variáveis específicas, e é dado por:

$$custo(c) = (custo\_esc_b(k-1) - custo\_esc_b(k)) * F(b)$$

onde k é o número de vértices representando candidatos locais ao bloco b presentes no grafo no momento do bloqueio. Observe-se também que candidatos locais considerados derramados não mais fazem parte do grafo de interferências em futuras tentativas de coloração. Cabe ao alocador gerar o código de derramamento.

A alocação de registradores durante a reorganização torna esta mais complicada. Pode ocorrer que após o escalonamento de parte da instruções, todas as prontas criem candidatos e que o limite de registradores físicos já tenha sido alcançado. Quando isto acontece, código de derramamento deve ser emitido. A emissão consiste em liberar imediatamente um registrador físico, através de um STORE, e introduzir vértices no grafo de dependências representando os LOADs. Os novos vértices (representando LOADs) devem preceder todas as instruções que utilizam o candidato local associado àquele registrador (criando novos candidatos).

A escolha do candidato a ser derramado baseia-se na altura do seu último uso no grafo de dependências; quanto mais próximo das folhas estiver seu último uso, menores as chances do escalonamento voltar a ser bloqueado.

Comentários

A técnica trabalha muito bem em máquinas com pequeno conjunto de registradores, baixa latência de acesso à memória e alta latência de instruções aritméticas. O tamanho dos blocos básicos também é fator determinante; quanto maiores, mais efetiva é a técnica. Estas condições são encontradas em programas científicos usando muitas instruções de ponto flutuante e executando em algumas máquinas RISC disponíveis. Nesse contexto o paralelismo necessário, para evitar a latência das computações, justifica a emissão de código de derramamento e conduz a reduções significativas no

tempo de execução dos programas; otimizações superiores a 35% foram obtidas para o Motorola 88000, em relação à aplicação em separado das duas otimizações; a média, contudo, ficou em 13%. Para o i860, que tem mais registradores e menores latências, a otimização média ficou em torno de 6%.

Há vários problemas com a abordagem. O pior deles é o custo; o reorganizador além de mais complicado é executado várias vezes. Não é claro como se dá a interação entre escalonamentos de várias classes de registradores (inteiros, ponto flutuante). A decisões sobre derramamento durante a reorganização também são complicadas. O escalonamento do LOAD que precede cada uso de um registrador derramado, se feito muito cedo, aumenta desnecessariamente o número de registradores em uso, podendo gerar mais derramamentos; se retardado demasiadamente reduz as oportunidades de reorganização de instruções que o sucedem.

6.6 Conclusões

Reorganização foi reconhecida desde cedo como um dos pontos importantes no aumento de desempenho de arquiteturas RISCs. A simplicidade dos *pipelines* contribui para que o processo possa ser executado em tempo satisfatório durante a compilação. Os resultados são estimulantes, sendo comuns reduções de 10% no tempo de execução de programas já submetidos às demais otimizações.

Motivados pela compatibilidade entre várias implementações de uma mesma arquitetura, projetistas têm procurado resolver dependências em *hardware*, mas a redução no tempo de ciclo proporcionada pela ausência deste tratamento ainda é forte o suficiente para motivar projetos de máquinas VLIW. Neste modelo, a resolução da maior parte dos conflitos que podem ocorrer em tempo de execução ainda é deixada a cargo do compilador.

Algoritmos que executam reorganização, em vários contextos, foram abordados neste capítulo, em particular, apresentou-se uma generalização de um método anterior que é capaz de trabalhar com muitas máquinas atualmente no mercado.

Uma outra conclusão é que os interrelacionamentos entre reorganização e alocação de registradores poderão se tornar mais importantes com o crescente interesse em técnicas que aumentam o tamanho dos blocos básicos; hoje, no entanto, são relevantes apenas em contextos específicos.

Capítulo 7

Experimentos e Experiências

"A great discovery solves a great problem, but there is a grain of discovery in the solution of any problem. Your problem can be modest; but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you can experience the tension and enjoy that triumph of discovery."
G. Polya

7.1 Introdução

No decorrer do trabalho vários pontos despertaram "curiosidades" com respeito à viabilidade de fazer algumas otimizações específicas para a arquitetura SPARC. Em cada um dos casos, o GCC foi usado como base para testar e implementar tais otimizações. Em geral, os resultados não foram significativos a ponto de permitir o largo uso das implementações. A exceção foi a implementação de um reorganizador de instruções que, além de totalmente operacional e integrado ao GCC, oferece melhorias significativas no tempo de execução dos programas onde aplicado. A implementação do reorganizador também fez possível a coleta de várias estatísticas sobre o uso de instruções na SPARC, algumas delas já apresentadas em capítulos anteriores.

Na próxima seção se discute, brevemente, o GCC. Uma investigação sobre a mudança na convenção de passagem de parâmetros em registradores é apresentada na seção 7.3. As duas seções que a seguem tratam de tentativas de alocação interprocedimental de registradores na arquitetura SPARC. Finalmente é apresentada a implementação de um reorganizador de instruções.

7.2 GCC

O GCC – *Gnu C Compiler* – é um compilador de linguagem C (ANSI) disponível para mais de 40 plataformas, entre máquinas e sistemas operacionais. GCC tem em torno de 100.000 linhas de código fonte e é totalmente escrito em C; também não possui qualquer código dependente de máquina. A sua estrutura não busca alta qualidade do código objeto para máquinas particulares, porém um vasto catálogo de otimizações é disponível, de modo que em alguns casos (por exemplo, SPARC) ele é competitivo com o compilador C do próprio fabricante. Seu objetivo principal, todavia, é portabilidade.

A versão do compilador a que se teve acesso carece de "otimizações RISC" mais elaboradas. Porém, o livre acesso ao seu código fonte permitiu que algumas estratégias de otimização fossem verificadas.

As subseções seguintes oferecem uma visão geral introdutória ao GCC. A próxima descreve a estrutura geral do compilador, segue-se uma introdução à estratégia de geração de código e sua interação com a migração do GCC para uma arquitetura específica. A última subseção apresenta as otimizações que o GCC implementa, com alguns comentários relevantes; esta lista serve como uma espécie de informação sobre as otimizações disponíveis em compiladores de produção atuais. Em algum instante, cada um dos assuntos abordados nas subseções esteve envolvido no processo de experimentação desenvolvido ao longo do trabalho.

7.2.1 Estrutura do Compilador

A Figura 7.1 mostra a estrutura de controle do compilador (DM é abreviação para Descrição da Máquina alvo). O *parser* é invocado uma única vez e à medida que o código de uma computação ou declaração é analisado, vai sendo convertido para uma estrutura de árvores sintáticas. A seguir as árvores são convertidas para RTL, *Register Transfer Language*, a verdadeira linguagem intermediária.

RTL é uma linguagem inspirada em Lisp, onde o operador vem antes dos operandos e estes por sua vez podem envolver mais operadores. Um programa é representado como uma lista duplamente encadeada de instruções RTL. A maior parte da compilação e praticamente todas as otimizações envolvem a representação RTL do programa, entretanto o estado de uma compilação não está totalmente representado na RTL.

7.2.2 Transporte de GCC e Estratégia de Geração de Código

O transporte[1] de GCC está intimamente ligado à sua estratégia de geração de código. Nos seguintes instantes da compilação de um programa características da máquina alvo estão envolvidas: geração da RTL, otimização e emissão de código de montagem.

GCC tem embutidas várias estratégias de geração da RTL para cada operação ou comando presente na estrutura de árvore. Por exemplo, uma expressão lógica pode ter código emitido de várias maneiras, sendo o critério escolhido em função de especificações RTL presentes na descrição de máquina. Um conjunto mínimo de tais especificações deve estar definido na máquina alvo, de outro modo, o compilador poderia não trabalhar. A emissão da RTL ainda incorpora vários outros detalhes, tais como: tamanho e ordenação dos *bytes* na palavra da máquina, alinhamento de dados e convenção de passagem de parâmetros.

A definição precisa dos registradores é importante durante a alocação (inicialmente, GCC supõe infinitos registradores simbólicos), entre as informações estão: número, tamanho, classe (inteiro, ponto flutuante) e categoria (preservado, volátil) de cada registrador, usos específicos (contador de programa, apontador do topo da pilha) e sobreposições entre registradores.

As especificações RTL também são usadas para a emissão de código de montagem. Elas são baseadas em padrões incompletos de RTL. Uma especificação é composta dos seguintes atributos, detalhados posteriormente:

- um nome de identificação da especificação;

[1] do inglês: *porting*.

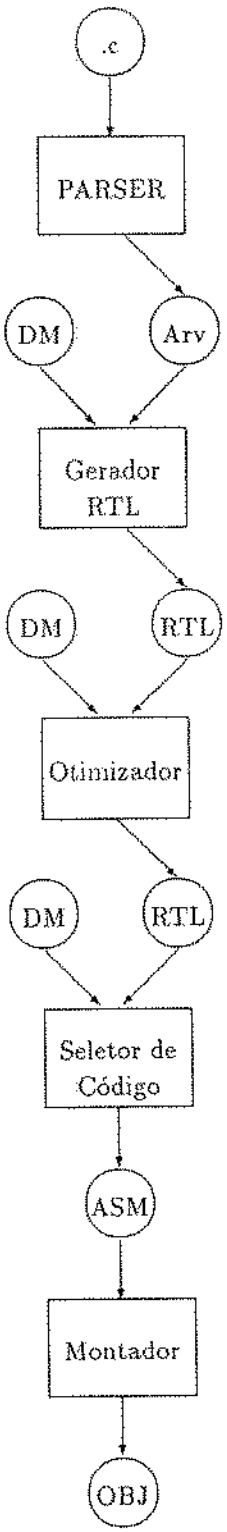


Figura 7.1: Estrutura do *Gnu C Compiler* – GCC.

- um padrão RTL onde alguns dos operandos são substituídos por nomes de funções;
- uma condição;
- um trecho de código escrito em C que é usado para emitir o código de montagem.

O nome da especificação serve como chave para determinar que estratégias de geração da RTL estão disponíveis.

O padrão RTL tem duas finalidades. Durante a geração de RTL, a especificação associada a determinado nome torna disponível uma estratégia de geração de RTL. Neste caso, o compilador emite o padrão presente na especificação, sendo os operandos (que a rigor não estão definidos) substituídos por aqueles de posse do gerador.

Para a emissão do código de montagem, cada instrução RTL é tentativamente casada com uma especificação, via o padrão RTL desta. Partes constantes dos padrões casam se são iguais. Alternativamente, a função que ocupa o espaço do operando na especificação é executada sobre o operando sendo casado, se a função retorna um valor verdadeiro o casamento acontece. Quando este casamento ocorre para todas as partes dos padrões, o código C associado à especificação é executado, emitindo o código de montagem correspondente. Casamentos são tentados na ordem que especificações são escritas em um arquivo de especificações, parando no primeiro sucesso.

A condição presente na especificação permite que ela seja válida apenas sob determinadas circunstâncias, por exemplo, uma opção de otimização particular.

O nome de uma especificação pode ser, eventualmente, nulo o que permite usar especificações apenas para o casamento de padrões. Isto é importante porque algumas otimizações criam ou mudam instruções RTL e é necessário reconhecê-las para emitir o código final. Padrões anônimos também podem ser úteis na emissão de código mais eficiente para situações específicas (*e.g.*, incremento).

Em resumo, transportar GCC equivale a criar as especificações e definir as características da máquina alvo. Outras destas características dizem respeito à forma de emitir código de montagem reconhecível pelo montador, por exemplo, nome dos registradores (internamente são conhecidos por números), forma correta de escrever constantes e endereços e diretivas que indicam segmentos de dados e/ou código.

7.2.3 Otimização

Conforme a Figura 7.1, a grande maioria das otimizações que o GCC faz é sobre a RTL. A lista abaixo sumariza as otimizações na ordem que são aplicadas.

1. *Constant folding*: resolve expressões envolvendo constantes. É a única otimização aplicada sobre o formato de árvores sintáticas.
2. Eliminação de recursão de cauda: a função não pode ter qualquer informação armazenada na pilha.
3. Inversão de malhas: construções repete-até, em geral, são mais eficientes que enquanto-faça, por isto o otimizador faz estas inversões.

4. Expansão de funções *inline*: os critérios para emitir código *inline* para uma função são o seu tamanho, número de parâmetros e ausência de recursão (exceto de cauda). Uma expansão à linguagem C implementada no compilador permite que o usuário especifique que uma função será *inline*.

5. Remoção de código não alcançável: código e rótulos inalcançáveis são removidos. Desde que até este ponto nenhuma análise de fluxo de dados foi feita, blocos básicos que formam subgrafos cíclicos no grafo de fluxo de controle ainda que inalcançáveis não são removidos neste estágio;

6. Otimização de desvios: elimina desvios para a instrução seguinte, desvios para desvios e aplica a regra 4 da Tabela 6.1.

7. Eliminação de subexpressões comuns: faz também propagação de constantes;

8. Movimentação de instruções invariantes em malhas: restringe-se a malhas naturais (não envolvendo *gotos*);

9. Análise de fluxo de dados: divide o programa em blocos básicos e calcula informação de vida das variáveis. Além disso, remove computações não utilizadas e código inalcançável;

10. Alocação local de registradores: Associa registradores físicos aos registradores simbólicos vivos em um único bloco básico;

11. Alocação global de registradores: Alocação de registradores aos demais registradores simbólicos;

12. *Reloading*: Emissão de código de derramamento e tratamento de assimetrias das várias máquinas onde o compilador pode ser executado.

As otimizações de 6 a 11 só são executadas sob requisição explícita. Se este não é o caso, o compilador faz uma alocação de registradores bastante ingênua.

7.3 Passagem de Parâmetros em Registradores de Ponto Flutuante

A convenção de passagem de parâmetros utilizada em compiladores para a SPARC emprega apenas registradores da UI (em número de seis), independentemente do tipo do parâmetro. No caso de valores de ponto flutuante observa-se que a ausência de comunicação direta entre registradores da UI e UPF introduz ineficiência no processo. A passagem de um parâmetro residente em um registrador de ponto flutuante (RPF), seja por alocação ou por se tratar do resultado de uma computação, envolve seu armazenamento em memória e posterior restauração para um registrador da UI dedicado à passagem de parâmetros. No procedimento chamado, o oposto acontece antes que o dado possa ser utilizado em computações.

A fim de verificar os efeitos da convenção padrão sobre o tempo de execução de programas cujas chamadas envolvem parâmetros de ponto flutuante, o GCC foi modificado para passar até três parâmetros de ponto flutuante em RPFs; mais seis (inteiros) seriam dedicados à passagem de parâmetros dos demais tipos.

Praticamente todos os programas fazem chamadas a bibliotecas e por este motivo, a passagem de parâmetros usando RPFs foi restrita a programas de um único módulo, onde procedimentos chamados, mas não definidos no módulo continuam a ter parâmetros sendo passados de acordo com a convenção padrão.

Avaliação

O único programa, entre os disponíveis, que pôde ser utilizado para avaliar o efeito da mudança na convenção de passagem de parâmetros foi **navega**. O seu tempo de execução foi tomado sob várias circunstâncias, conforme apresentado na Tabela 7.1. A primeira linha (de números) indica o tempo de execução quando registradores da UI são usados na passagem de "qualquer" parâmetro e segunda quando RPFs são empregados na passagem de valores de ponto flutuante.

Para compreender os resultados é necessário conhecer a estratégia de alocação de registradores de GCC. Um candidato que não cruza chamadas é prioritariamente alocado a um registrador volátil da classe a que pertence (*e.g*, ponto flutuante), mas na ausência de registradores voláteis na categoria, um registrador preservado é selecionado. O mesmo raciocínio não se aplica a candidatos que cruzam chamadas. Neste caso, um registrador preservado da classe é tentado, mas na ausência de um, registradores preservados de outras classes é que são escolhidos.

Na especificação GCC para a SPARC, registradores inteiros podem guardar valores de ponto flutuante. Como todos os RPFs são originalmente tratados como voláteis, todo candidato de ponto flutuante que cruza chamadas é associado a um registrador inteiro (e preservado) com todas as operações sobre ele sendo precedidas (sucedidas) de movimentação entre as classes de registradores (via memória). Os tempos na primeira coluna da Tabela 7.1 expressam a adoção deste critério, ou seja, não há registradores de ponto flutuante preservados, sendo registradores inteiros empregados para qualquer candidato que cruze chamadas.

	Registradores de ponto flutuante	
Parâmetros em:	preservados: 0 voláteis: 32	preservados: 16 voláteis: 16
Reg. Inteiros	52,38s	55,75s
Reg. P. Flutuante	52,37s	53,30s

Tabela 7.1: Efeito da convenção de passagem de parâmetros na SPARC.

Uma abordagem alternativa mantém candidatos (que cruzam chamadas) de ponto flutuante em RPFs voláteis, ao custo de gerenciá-los durante as chamadas. Mesmo existindo uma opção para GCC emitir código deste modo, a versão usada possui um *bug* e não gera resultados corretos.[2] Para forçar o uso de RPFs em candidatos de ponto flutuante cruzando chamadas, dividiu-se, então, o conjunto de RPFs igualmente entre voláteis e preservados (16 e 16). O tempos estão na segunda coluna da Tabela 7.1.

Os resultados são pouco conclusivos. É importante notar, entretanto, que só há vantagens em passar parâmetros em RPFs se os valores sendo passados já estão em RPFs. Esta observação é con-

[2]O problema é que GCC não consegue determinar corretamente os registradores que necessitam ser gerenciados (vivos) a cada nova chamada. Conseguiu-se resolver este problema, mas o código continuou com erro, que não pôde ser identificado.

firmada pelos números na segunda coluna da Tabela. Olhando de outro modo, isto também explica, parcialmente, o pior desempenho ao se dividir o conjunto de RPFs e continuar usando registradores inteiros para passagem de parâmetros. A outra razão para a queda de desempenho observada na primeira linha é que alguns procedimentos folha muito freqüentemente executados necessitam de muitos registradores; no primeiro caso existem registradores suficientes para alocação sem derramamentos e sem gerenciamentos; a diminuição no número de registradores voláteis forçou o uso de registradores preservados incorrendo em custo extra de gerenciamento. Os números praticamente idênticos na primeira coluna são provavelmente porque o número de valores de ponto flutuante, passados como parâmetro, que está em RPF e em registradores inteiros são aproximadamente iguais.

Diante da impossibilidade de experimentar o emprego de todos os registradores como voláteis, permitindo que eles sejam alocados a variáveis vivas durante chamadas (obviamente gerenciando-os), não se pode justificar o porquê da escolha da convenção adotada na SPARC. Possíveis razões são: 1) o custo de movimentação de RPFs é dependente da implementação da UPF; 2) o sistema de janelas elimina o custo de gerenciar registradores dedicados à passagem de parâmetros, pois usa diferentes registradores para parâmetros formais e efetivos; 3) experimentos confirmam [Cho88] que procedimentos muitas vezes retornam sem executar trechos de código responsáveis pelo uso da maior parte dos registradores, casos estes onde o gerenciamento devido aos registradores preservados é desnecessário.

7.4 Otimização do Uso de Janelas na SPARC

Baseados no baixo número de registradores necessários por procedimento e no alto custo de gerenciar estouros no sistema de janelas de registradores da SPARC, pensou-se em fazer com que vários procedimentos pudessem ser executados compartilhando registradores de uma mesma janela, o que teria o efeito de diminuir o número de acessos a memória decorrentes do gerenciamento do arquivo de registradores.

Para implementar a idéia seria necessário criar o grafo de chamadas estático e propagar usos de registradores, como em algum alocador interprocedimental descrito no capítulo 5. Sempre que o número de registradores de uma janela fosse esgotado, um novo conjunto seria requisitado, via uma instrução `save` no seu prólogo (vide discussão na página 67) do procedimento onde a exaustão ocorresse. De posse dos registradores disponíveis a cada procedimento, o alocador seria novamente invocado para atribuir definitivamente os registradores às variáveis.

Rumo à implementação, acoplou-se ao GCC um programa que coleta os dados necessários à construção do grafo de chamadas, bem como o número de registradores usados por cada procedimento. Análises destes dados revelaram que infelizmente o número de procedimentos onde a instrução `save` continua necessária é muito alto. No 001.gcc1.35, por exemplo, supondo janelas de 24 registradores, apenas 16% dos procedimentos não mais exigem `save`; acrescente-se que grande parte destes são folhas, onde a requisição de uma nova janela só é necessária se o número de registradores utilizados excede o número de registradores destinados à passagem de parâmetros, independentemente de qualquer informação interprocedimental.

Depois, se trabalhou em determinar quanto poderia ser economizado com a remoção total do gerenciamento de janelas. Estes números, para alguns dos *benchmarks*, foram apresentados no

capítulo anterior (Tabela 5.1) e representam muito pouco do tempo de execução.[3] Concluiu-se, portanto, que mesmo removendo todos os `save`, o que estava longe de ser verdade, seria pouco provável conseguir alguma redução significativa no tempo de execução dos programas.

A idéia foi finalmente abandonada por causa da complicada implementação. Exemplos de complicações são a passagem de parâmetros em registradores, que exige convenção diferente para procedimentos com e sem `save` no prólogo, e gerenciamento do registrador onde a SPARC põe o endereço de retorno após cada chamada.

7.5 Alocação Interprocedimental de Registradores de Ponto Flutuante

Alguns dos programas desenvolvidos a fim de verificar a viabilidade da idéia apresentada na seção precedente e, sobretudo, a experiência adquirida com a organização do GCC evidenciaram que, com pouco esforço, um protótipo de um alocador interprocedimental de RPFs poderia ser implementado.

A implementação é uma combinação das várias técnicas apresentadas no capítulo 5. A alocação é dividida em duas fases, ou em termos de implementação, duas execuções do GCC sobre o mesmo programa. Na primeira, os dados necessários à construção do grafo de chamadas estático e as necessidades de registradores de cada procedimento são coletados. Para determinar tais necessidades, o conjunto de RPFs é dividido igualmente em voláteis e preservados durante a fase de coleta.

A segunda fase faz a alocação propriamente dita. De posse da informação anteriormente obtida, o grafo de chamadas estático é construído e a informação de uso dos registradores propagada, gerando novas convenções que são então informadas convenientemente ao alocador intraprocedimental. As convenções calculadas fazem com que o alocador (intraprocedimental) trabalhe, transparentemente, seguindo critérios interprocedimentais, conforme detalhado adiante.

Durante a construção do grafo de chamadas estático, circularidades no grafo são eliminados marcando procedimentos que são possíveis alvos de chamadas recursivas ou indiretas. Estes procedimentos correspondem a pontos onde o contexto pendente no subgrafo do qual é raiz deve ser gerenciado. Este gerenciamento faz com que o procedimento se comporte como folha em relação aos seus predecessores.

Como sempre, a idéia geral do alocador interprocedimental é fazer com que procedimentos relacionados no grafo de procedimentos utilizem registradores diferentes. O algoritmo trabalha distribuindo registradores aos procedimentos no grafo de chamadas estático, partindo das folhas e prosseguindo de maneira ascendente, com a associação de registradores a um procedimento só sendo executada após a associação a todos os sucessores do referido procedimento.

Um registrador r associado a um procedimento P indica que r pode ser usado em P sem qualquer gerenciamento no prólogo (e no epílogo), podendo também cruzar chamadas sem incorrer em qualquer custo em P. Não se exige, entretanto, que r seja usado por P com exclusividade; qualquer predecessor de P no grafo de chamadas estático pode usar r, mas considerando-o como volátil; por outro lado, os sucessores de P também podem usar r desde que o considerem como preservado. A grosso modo estas são as idéias desenvolvidas na seção 5.3.4.

[3] Em programas recursivos, onde se espera maior profundidade da cadeia dinâmica de chamadas e, por conseguinte, maior custo de gerenciar janelas, a idéia apresentada não é aplicável, pois implicaria na destruição de contextos prévios armazenados nos registradores.

Uma heurística foi empregada para determinar o número de RPFs a associar a cada procedimento A, *NR(A)*. Sejam $P(A)$ e $V(A)$ o número estimado de registradores preservados e voláteis, respectivamente, necessários pelo procedimento A.

Para procedimentos folhas o número de registradores associados corresponde à totalidade dos registradores necessários pelo procedimento, incluindo voláteis e preservados, ou seja:

$$NR(A) = P(A) + V(A)$$

Fazendo assim garante-se que o comportamento das folhas é o melhor possível.

Dois critérios são utilizados para associar registradores a um procedimento A não folha: primeiro, o número de registradores que A poderá considerar como voláteis deve ser, no mínimo, igual a $V(A)$; com isto gerenciamentos extras associados com candidatos que cruzam chamadas são evitados. Note-se que os registradores associados aos sucessores de A no *grafo de procedimentos*[4], $Suc[A]$ já são vistos como voláteis em A, cabendo ao alocador interprocedimental complementar o número mínimo estimado, se for o caso. Segundo, metade da estimativa de registradores preservados necessários para A são associados a A; com isto, parte do gerenciamento associado a registradores preservados é evitado. Em resumo, para todos os procedimentos não folha A vale a fórmula:

$$NR(A) = \max(0,\ V(A) \ - \sum_{\forall B \in Suc[A]} NR(B)) \ + P(A)/2$$

Quando o número de registradores é exaurido, a alocação passa a seguir um critério intraprocedimental com todos os registradores considerados como voláteis.

O alocador tem sido usado com programas de um único módulo, mas pode ser facilmente estendido para trabalhar com vários módulos, que mantém um convenção intraprocedimental entre procedimentos não conhecidos no instante da compilação de cada módulo. Neste caso, os procedimentos chamados, mas não definidos no módulo, têm associado a si o conjunto de registradores voláteis da convenção intraprocedimental e procedimentos que podem ser chamados de outro módulo devem ser pontos de gerenciamento. Este último requerimento é o único, de fato, não implementado para o suporte a múltiplos módulos; sua implementação é trivial, no entanto, a construção da linguagem C para especificar que um procedimento não pode ser chamado de outros módulos é raramente usada, resultando na marcação da maior parte dos procedimentos como pontos de gerenciamento.

Alternativamente poder-se-ia adotar uma abordagem em duas fases generalizada, como aquela apresentada no capítulo 5, que também já está parcialmente implementada. O requerimento extra consiste em desenvolver um programa que construa o grafo de chamadas estático a partir de informações coletadas para vários módulos.

Avaliação

Novamente, o único programa que pôde avaliar a alocação de RPFs de maneira interprocedimental foi **navega**. Uma otimização de 2,5% foi verificada no seu tempo de execução. A redução no tempo, devida ao alocador interprocedimental, é resultado da diminuição do número de LOAD/STOREs. Medidas estáticas indicaram um redução percentual de 42,1% no número de LOADs devido a

[4] vide definição no início do capítulo 5.

variáveis escalares (ou 27,7% no total de LOADs). A redução estática no número de STOREs escalares foi 50,8% (ou 36,9% no total de STOREs).

A baixa resposta dinâmica à otimização medida estaticamente é provavelmente devida à alta latência das instruções de ponto flutuante quando comparadas às latências das instruções de acesso à memória.

7.6 Reorganização de Instruções

Conforme discutido em vários dos capítulos anteriores, reorganizar instruções tem efeito significativo no tempo de execução de programas em uma máquina RISC. Existem poucos trabalhos que investigam o desenvolvimento de reorganizadores de código integrados a compiladores transportáveis e, talvez em razão disto, GCC não possua mecanismos para especificação de características da máquina alvo que permitem otimizar o uso de *pipelines*.[5]

No contexto da SPARC, GCC é um compilador atrativo porque o compilador C oferecido pelo fabricante não implementa o padrão ANSI para a linguagem. Decidiu-se, então, construir um reorganizador a fim de melhorar a qualidade do código objeto gerado pelo GCC. O reorganizador está, agora, disponível como mais uma opção do compilador, totalmente integrada às demais.

7.6.1 Detalhes da Implementação

O reorganizador é executado como um passo adicional do compilador que aceita mnemônicos da SPARC e emite o mesmo código com ordenação (possivelmente) diferente. Para facilitar a implementação da reorganização, o formato do código emitido pelo GCC foi ligeiramente modificado através da inserção de marcas que guiam o reorganizador; de outro modo, seria necessário reconhecer muitas das diretivas (nem sempre bem documentadas) aceitas pelo montador. As marcas inseridas, no entanto, tomam a forma de comentários, tornando possível desligar a opção de reorganização e compilar o programa normalmente.

Uma seqüência de blocos básicos, correspondente normalmente ao código de uma função, é tratada de cada vez pelo reorganizador. Inicialmente é feita a análise léxica e sintática de cada uma das instruções no corpo da função, agrupando-as, a seguir, em blocos básicos.

O escalonador é então executado sobre cada bloco básico. O escalonamento de *slots* de desvios é feito logo após o escalonamento do bloco. Depois do escalonamento de todos os blocos, existem *slots* que não puderam ser preenchidos, então um passo adicional é executado para preenchê-los empregando instruções de outros blocos básicos dentro da mesma função; uma descrição mais detalhada deste processo é dada adiante.

O escalonador propriamente dito foi implementado em duas versões: uma que usa um algoritmo guloso com critério de *branch-and-bound* e outra que emprega a técnica apresentada na subseção 6.3.3.

A Máquina Alvo

Uma das primeiras dificuldades encontradas para tornar o reorganizador eficaz foi encontrar definições precisas sobre o comportamento dos *pipelines* da SPARC. Os manuais não têm qualquer

[5] Através de tratamento *ad hoc*, GCC implementa alguns casos de escalonamento de desvios.

informação sobre dependência de recursos e os custos (e latências) das instruções de ponto flutuante variam de implementação para implementação. Na ausência destas informações, utilizou-se os resultados dos experimentos de Irlam [Irl91] para contenção na memória e as latências de instruções de ponto flutuante do coprocessador MIPS R3010, contemporâneo da SPARC. Dentro do programa, este valores são associados a nomes (`#define` no jargão de C), de modo que podem ser mudados instantaneamente na disponibilidade de dados mais precisos.

Os números usados são sintetizados na Tabela 7.2, que é uma extensão da Tabela 3.1 e das recomendações na seção 3.5. Onde há conflitos, valem os números neste capítulo, desde que os apresentados anteriormente expressam situações ideais.

	UI		UPF	
INSTRUÇÃO	CUSTO	LATÊNCIA	CUSTO	LATÊNCIA
STORE (64 *bits*)	8	-	-	-
MOV-PF (64 *bits*)	2	-	2	0
ADD-PF (32 *bits*)	2	-	2	2
ADD-PF (64 *bits*)	2	-	2	2
SUB-PF (32 *bits*)	2	-	2	2
SUB-PF (64 *bits*)	2	-	2	2
MUL-PF (32 *bits*)	2	-	2	4
MUL-PF (64 *bits*)	2	-	2	5
DIV-PF (32 *bits*)	2	-	2	12
DIV-PF (64 *bits*)	2	-	2	19
OUTRAS-PF	2	-	2	2

Tabela 7.2: Custos e Latências usadas na reorganização de instruções.

O Grafo de Dependências

Experiências anteriores [War90] evidenciaram que grande parte do tempo de reorganização é devido à construção do DAG de código para cada bloco básico. A forma de construir o grafo pode também determinar um pós-processamento a fim de calcular os valores corretos para a altura e número de sucessores de cada vértice, usados como critérios de heurística.

A maneira mais simples de construir um DAG de código é partir da última instrução (na ordem original) e relacioná-la com todas as que a precedem e fazem referências aos mesmos dados. Este processo é aplicado à penúltima instrução e assim sucessivamente. Este método tem duas desvantagens: toma tempo de execução quadrático no número de instruções e pode gerar arestas transitivas excessivamente, tornando também mais lento o algoritmo escalonador, que tem tempo de execução linear no número de arestas (se ignoradas dependências de recursos).

Em vez disto, usou-se o seguinte método: inicialmente formou-se listas encadeando todas as definições de um mesmo recurso. Isto pode ser feito em tempo linear no número de instruções. A seguir, o bloco básico é novamente percorrido, da última para a primeira instrução, com cada instrução que usa um dado sendo relacionada à que o define prévia e futuramente. Não se argumenta que este esquema elimina arestas transitivas, mesmo porque tais arestas são até necessárias em alguns casos, vide por exemplo a Figura 7.2, mas as reduz significativamente.

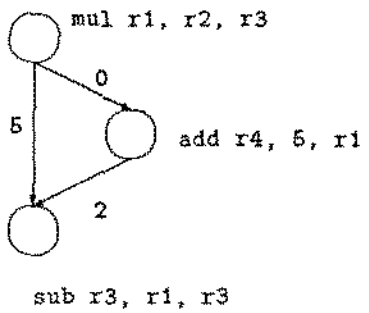


Figura 7.2: Benefício de arestas transitivas no grafo de dependências.

A aplicação estrita deste método serializa quaisquer referências à memória. Vários trabalhos, por exemplo [GM86], relatam que esta serialização dificulta o escalonamento. Desde que a maioria das referências a variáveis locais e parâmetros em código emitido pelo GCC é feita por um deslocamento constante relativo ao registrador apontando para o início do registro de ativação corrente, melhorou-se o processo de serialização de referências à memória: duas referências à memória nunca são serializadas se os endereços possuem deslocamentos constantes e distintos em relação a um mesmo registrador.

Finalmente algumas arestas adicionais são criadas para impedir que "instruções" que podem vir a ser convertidas em mais de uma em tempo de montagem, sejam escalonadas no *slot* de um desvio. No contexto da SPARC, a única "instrução" que pode ser convertida em duas é `set`, que move uma constante de *32 bits* para um registrador (lembre que instruções da SPARC só tratam constantes de até *22 bits*).

Tratamento de Slots de Desvios

Desvios são tratados usando o algoritmo apresentado na seção 6.3.3. No entanto, o *bit* de anulação disponível nas instruções de desvio permite que mais *slots* sejam preenchidos usando instruções do bloco básico que é possível alvo do desvio. A descrição de anulação está na seção 3.5.

A busca por uma instrução para execução condicional no *slot* do desvio só acontece quando uma instrução dentro do bloco não pôde ser conseguida. Se este é o caso, o bloco básico alvo do desvio é procurado entre os que compõem a função corrente.[6] Neste ponto todos os blocos básicos já foram escalonados. A primeira instrução escalonada no bloco básico alvo é então repetida no *slot* que finaliza o bloco corrente e o alvo do desvio modificado apropriadamente. Na presença de informações sobre a freqüência de execução de cada bloco básico, as recomendações da seção 3.5 poderiam ser aplicadas; por enquanto, não o são.

Nem sempre a primeira instrução no bloco básico alvo pode ser deslocada para o *slot*. Exemplos são instruções que possuem seus próprios *slots* e a instrução `set` já mencionada anteriormente. A efetividade desta abordagem em dois níveis é apresentada na próxima subseção.

Há uma alternativa ao uso do *bit* de anulação que pode ser inclusive mais efetiva: o emprego

[6] Usa-se um esquema de *hashing* para acelerar a busca.

da regra 6 da Tabela 6.1. Aplicá-la requer, entretanto, análise de fluxo de dados, que não foi implementada em tempo de reorganização.

7.6.2 Avaliação

A maneira ideal de verificar quão bom é um algoritmo heurístico consiste em observar a redução percentual de ciclos que ele proporciona em relação a um algoritmo ótimo. Esta avaliação foi tentada, mas como já observado, o algoritmo ótimo é exponencial e a presença de alguns blocos básicos grandes nos programas de teste inviabilizaram a tentativa, pois nestes casos o tempo de execução do algoritmo é inadmissível (observado em algumas amostras). As estatísticas coletadas, portanto, referem-se apenas à eficácia do algoritmo heurístico implementado.

Informações sobre redução no tempo de execução dos programas foram coletados e são apresentados na Tabela 7.3. Os tempos são em segundos. Conforme nota na página 76, 022.li não pôde ser executado.

	Tempo de execução (s)		
Programa	sem reorg	com reorg	otimização
001.gcc-1.35	101,4	95,3	6,0%
008.espresso	216,7	200,6	7,4%
023.eqntott	107,8	124,7	13,6%
navega	52,4	51,0	2,7%
reorg	17,5	16,1	8,0%
stanford	3,0	2,9	3,3%

Tabela 7.3: Redução no tempo de execução após reorganização.

Uma outra constatação foi que os efeitos da reorganização e da alocação interprocedimental de registradores de ponto flutuante no programa **navega** são cumulativos: a otimização total atingiu 5,0%.

Blocos Básicos

Normalmente, quanto maiores forem os blocos básicos maiores são as oportunidades de reorganização, pois muitas das instruções estão relacionadas com atividades paralelas executadas dentro do bloco. A organização das instruções dentro de um bloco básico também têm influência direta na viabilidade de usar algoritmos exponenciais. Os tamanhos médios dos blocos básicos (em instruções) para os *benchmarks* utilizados (com otimização ligada) estão na Tabela 7.4. Por uma restrição de implementação, instruções de chamada delimitam blocos básicos.

Eficácia do Escalonamento de Desvios

Uma vez esclarecido que *slots* são preenchidos de três modos, os números da Tabela 2.3 podem ser refinados. Os novos números na Tabela 7.5 indicam o percentual de *slots* preenchidos com instruções do próprio bloco básico (incondicionalmente), os que utilizam o *bit* de anulação e os preenchidas com nop.

PROGRAMA	NO. DE BBS	TAMANHO MÉDIO
001.gcc-1.35	34.334	3,61
008.espresso	8.033	3,89
022.li	3.260	3,09
023.eqntott	1.274	3,56
navega	201	10,56
reorg	1.043	4,43
stanford	457	4,93

Tabela 7.4: O Tamanho dos blocos básicos.

PROGRAMA	% DE DESVIOS	% DE SLOTS PREENCHIDOS		
		INCOND	COM ANUL	nop
001.gcc-1.35	23,7%	41,4%	43,6%	15,0%
008.espresso	21,4%	55,9%	32,9%	11,2%
022.li	27,0%	62,4%	25,6%	12,0%
023.eqntott	23,2%	55,2%	30,5%	14,3%
navega	7,6%	72,8%	21,0%	6,2%
reorg	18,7%	39,4%	31,6%	29,0%
stanford	14,6%	49,7%	21,4%	18,9%

Tabela 7.5: Detalhamento do preenchimento de *slots* de desvios.

7.6.3 Investigações

A fim de dar sentido prático a alguns números coletados pelo reorganizador tentou-se correlacionar a otimização estática do número de ciclos com a redução verificada no tempo de execução dos programas. A otimização estática de um programa corresponde à melhoria no número de ciclos medidos estaticamente, supondo que cada bloco básico é executado exatamente uma vez.

O coeficiente de correlação entre as otimizações estática e dinâmica é dado pela relação desta por aquela. O coeficiente é posteriormente usado para estimar a redução no tempo de execução proporcionada por determinado experimento a partir das medidas estáticas. A Tabela 7.6 apresenta os custos de escalonamento estático dos programas antes e depois da reorganização. O coeficiente de correlação está na última coluna da mesma Tabela.

A conjectura que pode ser feita é sobre quão mais de otimização no tempo de execução pode ser obtido se todos os *interlocks* fossem removidos. Para isto coletou-se a informação sobre o número de ciclos onde o escalonador não conseguiu encontrar um instrução útil para escalonamento, desperdiçando aquele ciclo. O valor foi subtraído do custo estático do escalonamento, resultando no que se denominou custo mínimo estático do escalonamento.

O custo mínimo foi comparado ao custo do escalonamento sem reorganização resultando na otimização estática máxima alcançável. Este número foi convertido a uma estimativa da redução dinâmica no tempo de chamada através da normalização pelo coeficiente de correlação. Esta estimativa corresponde ao percentual máximo de otimização que pode ser conseguida com reorganização. Os números estão na Tabela 7.7.

Programa	Custo estático		Otimização		
	sem reorg	com reorg	estática	dinâmica	coef_cor
001.gcc-1.35	210.099	194.175	7,6%	6,0%	0,8
008.espresso	51.426	46.651	9,2%	7,4%	0,8
023.eqntott	7.192	6.520	9,3%	13,6%	1,5
navega	8.457	7.464	11,7%	2,7%	0,2
reorg	7.324	6.885	6,0%	8,0%	1,3
stanford	4.032	3.551	11,9%	3,3%	0,3

Tabela 7.6: Redução no número de ciclos (estáticos) após reorganização.

Programa	Custo estático		Otimização	
	sem reorg	mínimo	estática	dinâmica estimada
001.gcc-1.35	210.099	180.806	13,9%	11,1%
008.espresso	51.426	44.022	14,1%	11,3%
023.eqntott	7.192	6.224	13,6%	20,4%
navega	8.457	5.960	29,5%	5,9%
reorg	7.324	6.448	12,0%	15,6%
stanford	4.032	3.300	18,2%	5,5%

Tabela 7.7: Estimativa da otimização máxima devida a reorganização.

Os resultados indicam que ainda há alguma margem de otimização a ser conseguida, no entanto, é impossível dizer quanto dos ciclos perdidos são, de fato, reaproveitáveis sem executar algoritmos exponenciais.

O Efeito de Eliminar Anti-dependências

Alguma integração entre alocação de registradores e reorganização de instruções pode ser feita durante a reorganização. Um método simples consiste em identificar anti-dependências dentro de cada bloco básico e mudar o registrador relativo à segunda instância por outro não utilizado no bloco básico. Este processo tem sido denominado de renomeamento[7] de registradores. Esta segunda instância do registrador poderia não estar viva na saída do bloco básico. De novo, a análise de fluxo de dados "errada" de GCC inviabilizou a idéia, que poderia ser inclusive melhorada em vários aspectos.

O que se fez então foi verificar com medidas estáticas o efeito de eliminar todas as anti-dependências devidas a registradores no grafo de dependências de dados. Ou seja, fez-se um renomeamento perfeito. A estimativa do efeito da otimização estão na Tabela 7.8 e mostra que resultados seriam, em princípio, modestos comparados à simples reorganização. Isto serve para demonstrar que técnicas muito complicadas (e lentas) de interação entre reorganização e alocação são pouco úteis a programas usuais.

[7] do inglês: *renaming*.

Programa	Custo estático		Otimização	
	sem reorg	reorg + renom	estática	dinâmica estimada
001.gcc-1.35	210.099	190.795	9,0%	7,2%
008.espresso	51.426	45.606	11,0%	8,8%
023.eqntott	7.192	6.396	11,1%	16,5%
navega	8.457	7.310	13,6%	2,7%
reorg	7.324	6.823	6,8%	8,8%
stanford	4.032	3.525	12,6%	3,8%

Tabela 7.8: Estimativa do renomeamento (perfeito) de registradores no tempo de execução.

O Paralelismo nos Programas de Teste

Uma última estatística coletada visa calcular o grau de paralelismo disponível nos programas. A Tabela 7.9 mostra o paralelismo médio por bloco básico (em instruções). O grau de paralelismo de um bloco básico é o número de instruções no bloco dividido pelo número mínimo de ciclos necessário para executá-las, supondo todas as instruções com custo unitário e sem latência, numa máquina com infinitos recursos.

A medida foi feita sob duas circunstâncias: considerando o código como emitido por GCC e fazendo renomeamento dos registradores para eliminar anti-dependências, conforme apresentado anteriormente.

Programa	sem renom	com renom
001.gcc-1.35	1,49	1,63
008.espresso	1,59	1,99
022.li	1,69	1,90
023.eqntott	1,53	1,81
navega	1,84	2,52
reorg	1,44	1,54
stanford	1,48	2,01

Tabela 7.9: Grau de paralelismo médio por bloco básico.

Os números confirmam experimentos anteriores que indicam uma baixa disponibilidade de paralelismo em blocos básicos, sendo necessárias outras técnicas para aumentá-lo. Observe-se que mesmo o mais alto número de instruções por bloco básico em navega não aumenta em muito o paralelismo, devido provavelmente a serializações nas instruções que referenciam a memória (42% do total).

O renomeamento dos registradores (isoladamente) parece ter pouco impacto no aumento do paralelismo.

7.6.4 Comentários

Durante a implementação do reorganizador, várias estratégias de implementação de determinadas partes do programa foram tentadas. Por exemplo, em um estágio inicial, a construção do grafo foi feita usando um algoritmo que verificava relacionamentos de cada instrução com todas as demais. Este método é quadrático. Embora o método usado operacionalmente seja assintoticamente melhor, os efeitos da mudança de estratégia não significaram reduções no tempo de execução do programa, principalmente porque a maior parte dos blocos básicos tem poucas instruções. Quanto às arestas transitivas geradas em demasia pelo primeiro método, observou-se que isto implicou pouco (negativamente) nos resultados dos escalonamentos.

Uma outra técnica de escalonamento de desvio foi considerada. Ela era mais complicada e baseia-se primordialmente em emitir o desvio (se ele estiver pronto) no momento em que o número de instruções por escalonar, excluindo o desvio, é igual ao número de *slots*. Os resultados foram similares à técnica descrita, embora possam ser piores se desvios de mais de um *slot* estivessem presentes.

Serializações de referências à memória ainda limitam bastante o trabalho do escalonador. Informações de sinonímia poderiam, eventualmente, ser muito úteis para melhorar a qualidade do código reorganizado.

Números estáticos são pouco significativos por si só. O objetivo da correlação de números estáticos com medidas dinâmicas é dar algum grau de validação às medidas puramente estáticas, porém, nada garante que o comportamento dinâmico seguirá as estimativas.

Observando o coeficiente de correlação entre otimizações estática e dinâmica para o programa **navega**, percebe-se que as informações utilizadas para latência das instruções de ponto flutuante muito provavelmente estão erradas. Isto é porque a redução estática é por demais otimista em relação ao que é realizado dinamicamente. Se latências maiores forem consideradas, a redução no número de ciclos estáticos proporcionada pela aplicação do reorganizador será menor e o coeficiente de correlação mais realista.

7.7 Conclusões

Embora o esforço de implementação tenha sido modesto (aproximadamente 5 mil linhas de código C), acredita-se que os resultados aqui obtidos servem em alguns casos como demonstração da viabilidade de algumas otimizações específicas de máquinas LOAD/STORE, ou como indicativos de onde a busca de otimizações não é atrativa (pelo menos para emprego em situações usuais). No primeiro caso se enquadram reorganização e, em menor escala, alocação interprocedimental. As demais tentativas refletem a segunda hipótese. Em qualquer dos casos, os *insights* providos pelas implementações serviram para elucidar algumas questões que, embora aparentemente óbvias, exigem resultados práticos para serem compreendidas; sob este ponto de vista, os resultados das experiências com implementações superaram as expectativas que as antecederam.

Capítulo 8

Conclusão

"There will be time, there will be time – time for you and time for me, and time yet for a hundred indecisions, and for a hundred visions and revisions before the taking of toast and tea."
T. S. Eliot

Dificilmente um modelo de arquitetura sai da pesquisa para o mercado tão rapidamente como aconteceu com as máquinas RISC. Nesta dissertação procurou-se apresentar o porquê deste fenômeno. Uma conclusão aparente é que o aumento de desempenho de um computador de uso geral é resultado direto da interação entre *hardware* e *software*. A ênfase de RISC é que este processo deve acontecer de forma transparente, tal que compilador, sistema operacional, arquitetura e *hardware* sejam partes de um único sistema. O sucesso do modelo RISC resulta exatamente da posição em que ele se interpõe no processo de conversão de um programa (escrito em uma linguagem de programação de alto nível) em resultados gerados pelo *hardware*.

Um exemplo da eficácia de RISC está na integração entre *hardware* e compiladores. A simplicidade e o pequeno número de instruções tornam possível simplificar o *hardware* envolvido na implementação da arquitetura, resultando em maior desempenho; a regularidade e primitivismo das instruções não só simplificam a geração final de código, como o tornam mais eficiente, tanto em seu próprio tempo de execução quanto na qualidade do código por ele gerado. É claro que a necessidade de otimização é mais crítica em RISC e isto, em um certo sentido, traz uma complicação a mais para o compilador.

A interação entre arquitetura e otimizadores tem sido particularmente interessante nas decisões sobre projeto de *pipelines*. Desde que compiladores trabalham bem no sentido de descobrir determinados relacionamentos entre instruções é possível simplificar o *hardware* e aumentar o desempenho do sistema. Um exemplo é o emprego de desvios atrasados. Em outras palavras, cada componente deve ser responsável por fazer o que sabe fazer melhor.

A evolução da tecnologia motiva soluções alternativas. Enquanto máquinas RISC oferecem maior desempenho para sistemas uniprocessados, os avanços da tecnologia permitem que, nos dias atuais, mais de um processador possa ser colocado em uma pastilha, sem aumento significativo no custo.[1] Resulta que o trabalho cooperativo de alguns processadores ainda que ineficientes pode gerar melhores resultados que um único processador eficiente. Diante deste quadro, projetistas se atêm agora na melhor maneira de explorar esta cooperação, ainda mantendo a eficiência de cada

[1] No sentido de custo de fabricação, o custo de projeto é evidentemente maior.

elemento de processamento. Foram discutidos aqui dois modelos que seguem este último princípio: VLIW e Superescalares.

A ênfase maior do trabalho, entretanto, foi no suporte que compiladores oferecem ao aumento de desempenho de máquinas RISC. Em verdade, pôde-se observar que o surgimento de RISC é baseado neste suporte. Explorou-se algumas técnicas bem estabelecidas para alocação de registradores e reorganização de instruções. Técnicas ainda em estágio experimental, como alocação interprocedimental de registradores e *pipeline* de *software*, podem ser fontes de novas decisões de projeto de arquiteturas e por esta razão também foram abordadas.

Na parte da tese associada a compiladores, o trabalho foi além do aspecto descritivo. Implementações foram feitas tendo objetivos, tanto operacionais (um reorganizador de código SPARC para o GCC) como investigativos (grau de paralelismo nos SPEC *Benchmarks*). Os resultados foram satisfatórios.

É importante mencionar que as conclusões deste trabalho não se encontram unicamente neste capítulo. Para ter uma noção melhor dos resultados, o leitor é fortemente recomendado a ler as seções **Conclusões** que finalizam cada um dos capítulos.

8.1 Contribuições

O propósito inicial da dissertação é ser um *survey* sobre aspectos de compilação associados a arquiteturas RISC. Para atingir o objetivo, optou-se por introduzir, em detalhes, arquiteturas RISC em geral, e SPARC em particular. Como já mencionado, a tese vai além de descrições fiéis de algoritmos propostos anteriormente. Na maior parte dos casos, o item Comentários esteve associado às descrições e, eventualmente, aumenta ou sugere modificações ao algoritmo básico proposto, a fim de melhorar ou esclarecer o seu funcionamento. Implementações validaram algumas destas técnicas.

A seguir estão relacionadas contribuições que este trabalho apresenta:

- A maior parte das questões associadas à integração de compiladores e arquiteturas RISC pode ser encontrada em um único trabalho;
- Números sobre SPEC *benchmarks* ainda são raros. As implementações possibilitaram a coleta de várias estatísticas para uma arquitetura particular;
- A organização do conjunto de registradores em janelas foi discutida amplamente; o seu engenhoso funcionamento foi esclarecido. Chegou-se também a estimar o custo de gerenciamento das janelas no tempo de execução de programas típicos, vide seção 5.4;
- Uma pequena modificação ao método de alocação de registradores proposto por Chaitin (seção 4.4.1) visando diminuir a quantidade de código de derramamento gerado foi sugerida;
- Uma generalização do conceito de bloqueio, também no método de Chaitin, permite a alocação de variáveis a registradores de tamanhos diferentes. Tal generalização permite, por exemplo, a alocação de registradores em máquinas vetoriais;
- As questões associadas com particularidades no uso de registradores tem tido pouca atenção na literatura. Não se argumenta que a seção que trata do assunto seja propriamente inédita, porém boa parte da apresentação não teve suporte em qualquer trabalho prévio;

- Uma sugestão foi incluída no método de Wall (seção 5.3.1) para melhorar a alocação local de registradores aos procedimentos no contexto de um algoritmo interprocedimental;
- Um algoritmo de reorganização de código foi proposto e implementado, tratando, inclusive, escalonamento de desvios de forma bastante uniforme.

Há também várias falhas:

- Embora tenha sido justificada, a ausência de um capítulo sobre seleção de código (em máquinas RISC) empobrece o trabalho. Alguém procurando por compilação para arquiteturas RISC poderia considerar interessante a presença do tema nesta dissertação, mesmo em sacrifício, por exemplo, da separação dos capítulos 3 e 7. O tamanho final da dissertação e o tempo necessário para explorar mais este assunto foram, em princípio, as razões da opção escolhida;
- A escolha de uma versão antiga do GCC como base para implementações limitou os horizontes de experimentos. Um exemplo está na inviabilidade de testar o mecanismo de *renaming* com propósitos práticos, conforme indicado na seção 7.6.3. Não está claro, entretanto, se novas versões resolveram os problemas;
- As estatísticas coletadas são sobre dados estáticos e, portanto, de pouco significado, em vários casos.

8.2 Trabalhos Futuros

Existem alguns pontos sobre os quais mais investigações ou trabalho podem ser feitos:

- O reorganizador pode ser integrado a algum utilitário de *profile*. Sabendo quantas vezes cada bloco básico é executado, o reorganizador pode ser alimentado e informações mais precisas sobre tempo de execução obtidas;
- Não parece muito complicado (re)escrever um alocador interprocedimental que trabalhe com programas de múltiplos módulos efetivamente. A questão é saber se os poucos registradores de ponto flutuante da SPARC justificam o esforço.
- O projeto de um alocador interprocedimental levantado no item anterior, quando combinado com o reorganizador alimentado por medidas dinâmicas dá margem a algumas investigações sobre alocação de registradores a programas inteiros. É possível, por exemplo, estimar a redução obtida no tempo de execução dos programas em função do aumento no número de registradores alocados seguindo o referido critério. Tais números podem ser importantes em decisões de projeto de arquiteturas;
- Uma implementação do método de Bernstein, seção 6.3.1, para reorganização de instruções poderia ser tentada, já que sob algumas restrições ele gera escalonamentos ótimos;
- O grafo de dependência pode ser usado para descobrir oportunidades de otimização *peephole*, por exemplo, de definições não utilizadas. Isto é útil quando o reorganizador é executado sobre código não otimizado.

- Medidas dinâmicas também são úteis para determinar a melhor maneira de escalonar *slots* de desvios usando instruções de outros blocos básicos e o *bit* de anulação.

Algum trabalho adicional também pode ser feito sobre algoritmo de reorganização aqui proposto. Sua aplicação a máquinas superescalares parece imediata e ainda mantém a mesma simplicidade. O preenchimento explícito de nops em VLIWs parece complicar o algoritmo e poderia ser fonte de investigações futuras.

Finalmente, existem mais horizontes para explorar no que diz respeito a técnicas que fazem reorganização considerando vários blocos básicos simultaneamente. O crescente interesse científico em máquinas superescalares motiva o estudo; o que foi apresentado neste texto é bastante simplificado.

Apêndice A

Implementações RISC - Características

Arquitetura	Ano lançamento	Formatos de instruções	Tamanho do Endereçamento	Tamanho da instrução
MIPS R2000/R3000	1986	3	32	32
Sun SPARC	1987	4	32	32
Motorola 88000	1988	6	32	32
Intel i860	1989	?	32	32
IBM RS/6000	1989	?	32	32

Tabela A.1: Características gerais.

Arquitetura	Registradores inteiros	Registradores de Ponto Flutuante
MIPS R2000/R3000	31 x 32 bits + r0 = 0	16 x 32 ou 16 x 64 bits
Sun SPARC	(janelas) 31 x 32 bits + r0 = 0	16 x 32 ou 16 x 64 ou 8 x 128 bits
Motorola 88000	31 x 32 bits + r0 = 0	os mesmos inteiros
Intel i860	31 x 32 bits + r0 = 0	30 x 32 ou 15 x 64 bits
IBM RS/6000[1]	32 x 32 bits	32 x 64 bits

Tabela A.2: Conjuntos de registradores

[1] O IBM RS/6000 tem oito registradores de código de condição que podem ser usados independentemente. O resultado de uma comparação no Motorola 88000 pode ser atribuído a qualquer um dos registradores e as instruções de desvio condicional podem avaliar a condição em todos eles. Esta facilidade aumenta a mobilidade de instruções, inclusive, para fora de malhas. Para resultados práticos no SPEC, vide [HO91]

Arquitetura	registrador + deslocamento	registrador + registrador	registrador + registrador *scaled*
MIPS R2000/R3000	x		
Sun SPARC	x	x	
Motorola 88000	x	x	x
Intel i860	x	x	
IBM RS/6000	x	x	

Tabela A.3: Modos de endereçamento

Bibliografia

[AJU77] Alfred V. Aho, Steven C. Johnson, and Jeffrey D. Ullman. Code generation with common subexpressions. *Journal of the ACM*, 24(1), January 1977.

[AN88] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7), June 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AZ89] T. L. Adams and R. E. Zimmerman. An analysis of 8086 instruction set usage in MS DOS programs. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 17(4), April 1989.

[BC91] Dileep Bhandarkar and Douglas W. Clark. Performance from architeture: comparing a RISC and a CISC with similar hardware organization. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 19(2), April 1991.

[BF90] Andreas V. Bechtolsheim and Edward H. Frank. Sun's SPARCstation 1: a workstation for the 1990s. *Proceedings of the COMPCON*, April 1990.

[BG89] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1), January 1989.

[BGM+90] David Bernstein, Martin C. Golumbic, Yishai Mansour, Ron Y. Pinter, Dina Q. Goldin, Hugo Krawczyk, and Itai Nahshon. Spill code minimization techniques for optimizing compilers. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(6), June 1990.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(6), June 1991.

[Bra91] David Gordon Bradlee. *Retargetable instruction scheduling for pipeline processors*. PhD thesis, University of Washington, 1991.

[CAC+81] Gregory Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[Cam91] Alda Campos. RISC é só *marketing*: uma solução para um problema que não existe mais. *Revista Mundo Unix*, June 1991.

[CH84] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1990.

[Cha82a] Gregory Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 17(6), June 1982.

[Cha82b] A. E. Charlesworth. A approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 15(9), September 1982.

[CHK86] Deborah Coutant, Carol Hammond, and John Kelley. Compilers for new generation of Hewlett-Packard computers. *Hewlett-Packard Journal*, pages 4–18, January 1986.

[Cho88] Fred C. Chow. Minimizing register usage penalty at procedure calls. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7), June 1988.

[CIEDJK85] Robert P. Colwell, C. Y. Hitchcock III, H. M. B. Sprunt E. D. Jensen, and C. P. Kollar. Computer, complexity, and controversy. *IEEE Computer*, 18(9), September 1985.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(6), June 1991.

[CKDK91] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An analysis of MIPS and SPARC intruction set utilization on the SPEC benchmarks. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 19(2), 1991.

[CNO+88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. P. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, c-37(8), 1988.

[Den78] Peter J. Denning. Computer architecture: Some old ideas that haven't quite made it yet. *Communications of the ACM*, 0(0), March 1978.

[Dit82] David R. Ditzel. Register allocation for Free: The C machine stack cache. *International Conference on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 10(2), March 1982.

[DLSM81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, c-30(7), July 1981.

[DV87] Jack W. Davidson and Richard A. Vaughan. The effect of instruction set complexity on program size and memory performance. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 15(5), October 1987.

[Fis81] Joseph A. Fischer. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, c-30(7), July 1981.

[Gas89] Franco Gasperoni. Compilation techniques for VLIW architectures. Technical report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, 1989.

[GHPR88] Thomas R. Gross, John L. Hennessy, Steven A. Przybylski, and Christopher Rowen. Measurement and evaluation of the MIPS architecture and processor. *ACM Transactions on Computer Systems*, 6(3), August 1988.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architeture. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), June 1986.

[Gro90] Gregory F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), January 1990.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Hen84] John L. Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, c-33(12), December 1984.

[HG83] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3), July 1983.

[HKMW66] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *Journal of the ACM*, 13(1), January 1966.

[HO91] C. Brian Hall and Kevin O'Brien. Performance characteristics of architectural features of the IBM RISC System 6000. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 19(2), April 1991.

[Irl91] Gordon Irlan, 1991. Personal communication.

[KM89] Les Kohn and Neal Margulis. Introducing the intel i860 64-bit microprocessor. *IEEE-Micro*, August 1989.

[Kow83] Tomasz Kowaltowski. *Implementação de Linguagens de Programação.* Guanabara Dois, 1983.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language (ANSI C).* Prentice Hall, 1988.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7), June 1988.

[Lee89] Ruby Lee. Precision architecture. *IEEE Computer*, 22(1), January 1989.

[LH84] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), June 1984.

[LKB91] Roland L. Lee, Alex Y. Kwok, and Fayé A. Briggs. The floating point performance of a superescalar SPARC processor. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 19(2), April 1991.

[LS84] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1), January 1984.

[Man89] Udi Manber. *Introduction to Algorithms: A creative approach.* Addison-Wesley, 1989.

[Mel89] Charles Melear. The design of the 88000 RISC family. *IEEE-Micro*, April 1989.

[Muc90a] Steven S. Muchnick. Compiling for RISC-based systems. SIGPLAN'90 RISC Compilers Tutorial, 1990.

[Muc90b] Steven S. Muchnick. The Sun compiling system. Technical report, Sun Microsystems, 1990.

[Nav90] Philippe Navaux. *Processadores pipeline e processamento vetorial.* VII Escola de Computação, 1990.

[OG90] R. R. Oehler and R. D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1), January 1990.

[Pat85] David Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(6), June 1991.

[PS82] David Patterson and Carlo Séquin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, September 1982.

[PW89] Richard S. Piepho and William S. Wu. A comparison of RISC architectures. *IEEE-Micro*, August 1989.

[Rad82] George Radin. The 801 minicomputer. *International Conference on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 10(2), March 1982.

[SH89] Peter S. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1), January 1989.

[Sny84] Lawrance Snyder. Supercomputers and VLSI: the effect of large-scale integration on computer architecture. In Marshall C. Yovits, editor, *Advances in Computers*, volume 23. February 1984.

[SO90] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 25(6), June 1990.

[Sou91] Galileu Batista de Sousa. Otimização de código em RISC. Proposta de Dissertação de Mestrado, DCC-UNICAMP, May 1991.

[SPE91] SPEC. SPEC newsletter. 3(1), April 1991.

[Sta89] Richard M. Stallman. *Using and porting GNU CC*. Free Software Foundation, Inc., 1989.

[Sun87] Sun Microsystems, Mountain View. *The SPARC architecture Manual*, 1987.

[Tan78] Andrew S. Tanenbaum. Implications of structured programming for machine architecture. *Communications of the ACM*, 21(3), March 1978.

[Tan90] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, third edition, 1990.

[TL90] Tadao Takahashi and Hans K. E. Liesenberg. *Programação orientada a objetos*. VII Escola de Computação, 1990.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), January 1967.

[TS83] Yuval Tamir and Carlo Séquin. Strategies for managing the register file in RISC. *IEEE Transactions in Computers*, c-34(11), November 1983.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The theory and practice of compiler writing*. McGraw-Hill, 1985.

[Wal86] David W. Wall. Global register allocation at link time. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), June 1986.

[Wal88] David W. Wall. Register windows vs. register allocation. *ACM SIGPLAN Conference on Programming Language Design and Implementation. SIGPLAN Notices*, 23(7), June 1988.

[Wal91] David W. Wall. Limits of instruction-level parallelism. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 19(2), April 1991.

[War90] Henry S. Warren Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), January 1990.

[Wie82] Cheryl A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. *International Conference on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 10(2), March 1982.

[Wir87] Niklaus Wirth. Hardware organization for programming languages and programming languages for hardware architetures. *Symposium on Architectural Support for Programming Languages and Operating Systems, SIGARCH Computer Architecture News*, 15(5), October 1987.

[Wul81] Willian A. Wulf. Compilers and computer architecture. *IEEE Computer*, 14(7), July 1981.