Alocação Global de Registradores de Endereçamento Usando Cobertura do Grafo de Indexação e uma Variação da Forma SSA

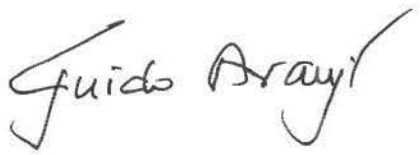
Marcelo Silva Cintra

Dissertação de Mestrado

Alocação Global de Registradores de Endereçamento Usando Cobertura do Grafo de Indexação e uma Variação da Forma SSA

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcelo Silva Cintra e aprovada pela Banca Examinadora.

Campinas, 18 de abril de 2000.



Prof. Dr. Guido Costa Souza de Araujo
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Alocação Global de Registradores de Endereçamento Usando Cobertura do Grafo de Indexação e uma Variação da Forma SSA

Marcelo Silva Cintra[1]

Abril de 2000

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araujo (Orientador)
- Prof. Dr. Roberto da Silva Bigonha
  Departamento de Ciência da Computação-UFMG
- Prof. Dr. Tomasz Kowaltowski
  Instituto de Computação - UNICAMP
- Prof. Dr. Arnaldo Moura (suplente)
  Instituto de Computação - UNICAMP

[1]**Financiado pela PRPG-CAPES(6 meses) e pela FAPESP processo: 98/06225-2**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Cintra, Marcelo Silva

C493a Alocação global de registradores de endereçamento usando cobertura do grafo de indexação e uma variação da forma SSA / Marcelo Silva Cintra – Campinas, [S.P. :s.n.], 2000.

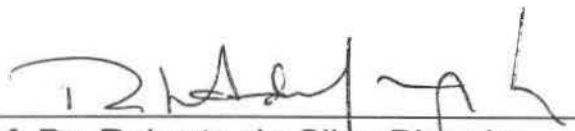
Orientador : Guido Costa Souza de Araujo

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Linguagem de programação (Computadores). 2. Compiladores (Computadores). 3. Arquitetura de computadores. I. Araujo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 18 de abril de 2000, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Roberto da Silva Bigonha
DCC-UFMG

Prof. Dr. Tomasz Kowaltowski
IC-UNICAMP

Prof. Dr. Guido Costa Souza de Araújo
IC-UNICAMP

Resumo

O modo de endereçamento indireto é o modo mais utilizado para acessos a arrays em programas que executam em arquiteturas CISC dedicadas. A razão para isto é que o endereçamento indireto permite o cálculo rápido de endereços usando instruções curtas. Este trabalho propõe uma solução para o problema de alocação de registradores de endereçamento para referências a elementos de arrays em laços, utilizando modo de endereçamento indireto combinado com auto-incremento. O resultado é um algoritmo que minimiza o número de registradores de endereçamento e instruções de redirecionamento requeridas por um programa. Este trabalho propõe uma extensão, para o caso multidimensional, de trabalhos anteriores baseados na cobertura do *Grafo de Indexação*(IG). Este trabalho propõe ainda um algoritmo de alocação global baseado em uma variação de *Static Single Assignment Form* e uma heurística para a redução do número de registradores requeridos pela cobertura do IG. Um compilador otimizante pertencente à Conexant Systems Inc. é utilizado para testar estas idéias. Resultados experimentais, usando programas reais, mostraram uma melhoria de desempenho de 11.3% no tempo de execução quando comparado com uma técnica de coloração baseada em prioridade. Devido ao impacto da alocação de registradores na geração de código, esta técnica pode melhorar substancialmente o tamanho do código gerado, reduzindo a dissipação de energia e aumentando o desempenho do sistema. Estas características são extremamente desejáveis para o projeto de computadores portáteis modernos.

Abstract

Indirect addressing is by far the most used addressing mode in programs running in embedded CISC architectures. The reason is that it enables fast address computation combined with short instructions. This work proposes a solution to the problem of allocating address registers to array references within loops, when using indirect addressing combined with auto-increment. The result is an algorithm that minimizes the number of address registers and redirect instructions required by a program. It extends previous work using *Indexing Graph*(IG) covering to the multidimensional case, and proposes a global allocation algorithm based on a variation of *Static Single Assignment Form.* This work also presents a heuristic that aims at reducing the number of address registers required by the covering of the IG. An optimizing production compiler from Conexant Systems Inc. is used to test the approach. Experimental results, using real world-programs, showed an 11% performance improvement when compared to a priority-based register coloring technique. Because of the impact of register alocation in code generation, this technique can substantially improve code size, power dissipation and performance, without increasing cost. These are very desirable features for the design of modern portable computers.

Aos meus queridos pais e irmãos.

Feliz aquele que levanta pela manhã
e acredita que pode fazer o mundo ser um pouco melhor,
por seu esforço, por sua dedicação e por seu trabalho.
Feliz aquele que acredita num amigo, na vida, no homem e em Deus.

Agradecimentos

A FAPESP e a PRPG-UNICAMP pelo suporte financeiro.

Ao meu orientador, Prof. Guido Araujo, pela amizade, pelo incentivo, pela confiança depositada, pelos valiosos comentários em nossas reuniões e pelas oportunidades que me proporcionou.

Aos funcionários e aos professores do Instituto de Computação.

Aos professores do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, que me apoiaram antes e durante o decorrer deste trabalho, em especial aos professores Ronaldo, Leandro, Marcelo Siqueira, Fabio e Edson.

A todos os amigos, que comigo ingressaram no programa de mestrado de 1998, pelos bons momentos. Em especial ao Nathan e Leonardo pelo apoio e pelo ótimo convívio que tivemos durante este período.

À Conexant Systems Inc., pelas ferramentas cedidas e pela oportunidade de estágio, o que foi muito importante para a concretização deste trabalho. Em especial ao Marek e ao Rajat pela orientação e esclarecimentos.

Aos amigos Amaury e Said, pela amizade e incentivo constante durante todo o desenvolvimento deste trabalho.

A todos os amigos formandos do curso de Ciência da Computação de 1997 da UFMS, que sempre me incentivaram.

Aos meus pais, Jair e Maria, e aos meus irmãos, Jean e Paulinho, a quem sou muito grato pelo apoio, pelos conselhos, pela preocupação, e acima de tudo pelo amor.

A Deus, ao meu anjo da guarda e aos espíritos protetores, que sempre estiveram ao meu lado iluminando os meus passos.

O meu muito obrigado.

MARCELO S. CINTRA

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

High thoughts must have high language.

Aristophanes

1.1 Motivação

Um dos problemas mais importantes em compilação é aquele de se atribuir registradores às variáveis de um programa. Este problema é conhecido como *Alocação de Registradores*. Encontrar uma atribuição ótima para os registradores é difícil. Por alocação ótima entende-se aquela em que maximiza o uso dos registradores no código resultante, minimizando o número de acessos à memória. Matematicamente, o problema é *NP-completo* [11]. O problema é adicionalmente complicado porque algumas arquiteturas, e em alguns casos o sistema operacional, exigem que certas convenções de uso dos registradores sejam observadas [1].

A importância deste problema advém da grande diferença entre os tempos de acesso aos registradores do processador e à memória em um computador moderno [19]. Técnicas eficientes de alocação de registradores geralmente resultam em melhorias consideráveis no desempenho de programas [1, 29].

Alocação de registradores é um problema importante na geração de código e tem sido extensivamente estudado em [9, 7, 10, 18]. Estas soluções são baseadas em coloração de grafos e são utilizadas em vários compiladores de produção. Outras pesquisas têm considerado a iteração da alocação de registradores com escalonamento de instruções na geração de código para máquinas RISC [17, 6], e a alocação de registradores entre procedimentos [8].

Este trabalho trata o problema de alocação de registradores de endereçamento em processadores que executam programas dedicados (ou embutidos). Esta classe de processadores inclui desde máquinas CISC comerciais (ex. Motorola 68000) até Processadores de Sinal Digital (*DSPs*[1]) (ex. ADSP 21000), que correspondem a uma parcela considerável dos processadores produzidos todo ano. Esses processadores dedicados estão sendo amplamente utilizados em sistemas de processamento de audio/vídeo e telecomunicações. Além disso, têm sido alvo de várias pesquisas, tanto industriais como acadêmicas. Este esforço de pesquisa vem sendo motivado pelo crescimento vertiginoso do mercado de sistemas de computação portátil nos últimos anos. Estimativas da indústria [14] prevêem que em cinco anos o mercado de sistemas de computação portátil irá ultrapassar o mercado de computadores *desktop*, um crescimento anual de cerca de 40%, muito maior que os 12% anuais esperados para o mercado de computadores *desktop*. Isto é um reflexo de uma mudança de paradigma em Computação, no qual a máquina está se especializando ao perfil e mobilidade do usuário final.

Dispositivos portáteis têm que obedecer fortes limitações tais como custo/tamanho reduzidos e baixa dissipação de potência, uma vez que são projetados para mercados de larga escala muito competitivos. Por este motivo, eles são normalmente construídos utilizando-se processadores dedicados de baixo custo e pequenas quantidades de memória. Tipicamente, o tamanho do código de programas dedicados é pequeno, geralmente menor que 4 MB de memória ROM, e deve ser executado com alto desempenho e ser extremamente otimizado, reduzindo assim o tempo de execução, e consequentemente o consumo de energia pelo processador, o que é de grande importância nestes dispositivos, uma vez que muitos são movidos à bateria. Devido às restrições de tamanho e desempenho, programas embutidos eram tradicionalmente escritos em linguagem de máquina. No entanto, o aumento do tamanho e complexidade das aplicações executando nestes sistemas vem tornando difícil a manutenção e desenvolvimento destes programas, o que tem motivado o desenvolvimento de compiladores otimizadores para estes [31, 23, 30, 22].

O uso de técnicas de geração de código convencionais geralmente produz código ineficiente para esta classe de processadores, uma vez que a sua arquitetura possui muitas limitações e irregularidades resultantes da necessidade de se reduzir custo e melhorar desempenho. Estas limitações se originam de um conjunto de registradores não homogêneo, um número pequeno de registradores e unidades funcionais especializadas, conectividade restrita no caminho de dados(datapath), e modos endereçamento especializados [28].

O cálculo de endereços constitui uma grande parcela do tempo de execução na maioria dos programas. Endereçamento pode representar mais de 50% dos bits de um programa e 1 a cada 6 instruções em um programa típico de propósito geral [20]. Estes números

[1] Do original em inglês: *Digital Signal Processors*

são provavelmente maiores no caso de um programa dedicado, dado que a maioria dos operandos são elementos de *arrays*. Com o intuito de acelerar a computação de endereços, muitos processadores DSPs/CISCs oferecem modos de endereçamento especializados. Um exemplo típico é o modo de auto-incremento (decremento), no qual um registrador é incrementado (decrementado) depois que uma referência à memória é realizada. Este modo está presente no conjunto de instruções de todos processadores DSPs e de muitos CISCs, e permite que instruções sejam codificadas em menos bits do que em outros modos, resultando em um código menor. Como consequência disto, o endereçamento indireto é amplamente preferido por programadores e projetistas de processadores dedicados.

Uma outra razão pela qual modos de auto-incremento (decremento) são muito utilizados advém dos tipos de dados presentes em programas dedicados. Na maioria destes programas, uma simples referência à memória é suficiente para se acessar um dado típico (ex. inteiro), de tal forma que o incremento ou decremento de um registrador de endereçamento é tipicamente a única operação requerida para computar o endereço do próximo dado na memória.

1.2 Trabalhos Relacionados

Registradores de endereçamento combinados com modo de auto-incremento (decremento) são tipicamente utilizados por um compilador com dois objetivos. Primeiro, permitir leitura/escrita de variáveis automáticas alocadas na pilha de execução. Segundo, permitir o acesso eficiente aos elementos de *arrays* em laços do programa. Neste último caso, o uso de auto-incremento (decremento) no acesso a elementos de arrays pode resultar em uma redução considerável no tempo de execução de um programa, dado que a maior parte deste é tipicamente dispendido em laços interiores.

O problema de se determinar o layout das variáveis automáticas na pilha, de forma a maximizar o uso de operações de auto-incremento (decremento) durante o acesso às mesmas, é conhecido como *Simple Offset Assignment* (SOA). SOA pode ser formulado da seguinte maneira. Assuma a existência de um único registrador de endereçamento `ar` que será utilizado exclusivamente para acessar (ler ou escrever) todas as variáveis automáticas de um procedimento. Determine a atribuição de deslocamentos às variáveis automáticas (em relação ao início da pilha) de modo a minimizar o número de instruções necessárias a atualização de `ar`. Observe que, se duas variáveis automáticas a e b são acessadas consecutivamente, e a posição ocupada pelas mesmas na pilha também é consecutiva, então auto-incremento (decremento) pode ser utilizado para redirecionar `ar` de a para b . Por outro lado, se a e b não ocupam posições consecutivas, uma *instrução de redirecionamento* é necessária para somar (subtrair) a distância entre estas variáveis na pilha.

SOA foi originalmente formulado e resolvido por Bartley em [4]. Bartley chama cada referência a uma variável na pilha de execução de acesso, e a ordem dos acessos dentro de um programa de *seqüência de acessos*. Acessos são associados a vértices de um grafo G, de modo que existe uma aresta (a, b) entre dois acessos a e b, se a (b) segue b (a) na seqüência de acessos. Cada aresta (a, b) em G é anotada com um custo 0, se auto-incremento (decremento) pode ser utilizado para redirecionar `ar` de a para b, ou 1 caso contrário. Note que um caminho que cobre todos os acessos de G (i.e. um *caminho hamiltoniano*) tem um custo correspondendo ao número de instruções necessárias a atualização de `ar`. Assim sendo, a solução para SOA requer se determinar o caminho hamiltoniano em G que possui custo mínimo.

Bartley mostrou [4] que esta solução para SOA resultava em uma otimização que melhorava consideravelmente o desempenho das instruções de acesso a variáveis automáticas na pilha de execução de um programa. Posteriormente, Liao et al [27] propuseram uma solução alternativa para SOA , provaram que SOA é um problema NP-difícil e estudaram heurísticas para uma generalização deste, que eles chamaram de *General Offset Assignment* (GOA). GOA é uma extensão de SOA em que um número arbitrário k de registradores de endereçamento são utilizados para acessar a pilha. Liao apontou que GOA pode ser adequadamente resolvido usando uma partição dos vértices de G em k sub-conjuntos, reduzindo assim GOA a k problemas SOA separados. Algoritmos mais eficientes para SOA foram propostos posteriormente em [4, 33, 34]. Uma heurística que melhora a partição dos acessos em GOA pode ser encontrada em [25]. Uma generalização destas soluções, que leva em conta outras arquiteturas foi estudada em [24]. Nesta abordagem, foram considerados registradores de endereçamento que aceitam auto-modificação através da adição/subtração de registradores de *offset*. Uma análise detalhada sobre os modos de endereçamento e a eficiência das unidades de endereçamento em arquiteturas DSPs pode ser encontrada em [32].

Conforme dito anteriormente, o acesso às variáveis automáticas de um programa é apenas um dos usos que o compilador faz dos registradores de endereçamento. Outra utilidade destes registradores ocorre durante a referência aos elementos de um array dentro dos laços de um programa. O problema de se alocar registradores de endereçamento às referências de um array, de modo a minimizar o número de registradores necessários ao endereçamento é conhecido como *Array Index Allocation* (AIA). Apesar das semelhanças, AIA e SOA/GOA são problemas diferentes que exigem soluções distintas. A solução para problemas como SOA/GOA visa basicamente determinar o layout de variáveis na memória, de modo a minimizar o uso de instruções de atualização. Nestes casos, o layout das variáveis na memória (i.e. pilha) depende da seqüência com que as variáveis são acessadas no programa. Observe que, se as variáves fossem elementos de um array o

problema tornar-se-ia mais difícil, dado que o reordenamento do layout de um array em memória traz consequências que podem afetar diretamente o desempenho de um programa (ex. uma mudança na ordem com que linhas de cache são acessadas) . Além disto, a ordem em que ocorrem as referências aos elementos do array somente é definida ao longo de várias iterações do laço dentro do qual estão as referências, tornando a solução deste problema ainda mais difícil. Por esta razão, qualquer solução para AIA deve garantir a manutenção do layout do array em memória, diferente da solução para SOA/GOA.

Araujo et al [3] originalmente propôs e resolveu o problema AIA [3]. Araujo associa um vértice a cada *referência* de um array dentro de um laço. Existe uma aresta (a, b) entre duas referências a e b se o módulo da diferença entre as expressões que são os índices das referências a e b for menor ou igual a 1. Observe que isto ocorre quando as referências a e b estão em posições consecutivas na memória, o que por conseguinte implica que eles podem usar auto-incremento (decremento) para redirecionar o registrador de endereçamento entre eles. O grafo resultante desta formulação é conhecido como *Indexing Graph* (IG). Um caminho no IG corresponde a alocação do mesmo registrador de endereçamento a uma seqüência de referências de array. A solução para AIA usando este algoritmo visa portanto minimizar o número de caminhos que cobrem o IG.

Posteriormente, Gebotys [16] estendeu esta abordagem, através da solução de um problema de fluxo em redes, com o intuito de minimizar o número de instruções de redirecionamento necessárias dentro do laço. Leupers e Marwedel [26], por sua vez, incorporaram o trabalho de Araujo a uma estratégia tipo *branch-and-bound* visando encontrar uma solução exata para AIA.

1.3 Objetivos

A solução para o problema de alocação de registradores proposto em [9, 7, 10, 18] não tira proveito das características de auto-incremento(decremento) de registradores, caso o modo de endereçamento de auto-incremento(decremento) esteja presente na arquitetura. Este trabalho propõe então duas técnicas para a alocação de registradores de endereçamento em arquiteturas que possuam modos de endereçamento indireto baseado em auto-incremento (decremento). Estas técnicas realizam a alocação de registradores de endereçamento para referências a elementos de arrays que ocorrem no interior de laços e procuram, sempre que possível, tirar vantagem das propriedades de auto-incremento(decremento) existentes nestes registradores, tornando assim o código no interior de um laço mais compacto e, conseqüentemente, mais eficiente.

A primeira técnica apresentada (Capítulo 2) é baseada na cobertura do IG usan-

```
(1)   for (i = 1; i < N-1; i++) {
(2)     avg = a[i] >> 2;
(3)     if (i % 2){
(4)       avg += a[i-1] << 2;
(5)       a[i] = avg * 3;
(6)     }
(7)     if (avg < error)
(8)       avg -= a[i+1] - error/2;
(9)
(10)
(11)  }
(12)
```

(a)

```
p = &a[1];
for (i = 1; i < N-1; i++) {
   avg = *p++ >> 2;
  if (i % 2) {
    p += -2;
    avg += *p++ << 2;
    *p++ = avg * 3;
  }
  if (avg < error)
     avg -= *p - error/2;
}
```

(b)

Figura 1.1: (a) Fragmento de código. (b) Código modificado que permite a alocação de um registrador para todas referências.

do caminhos. Esta técnica é uma extensão do trabalho de Araujo et al em [3] para arrays multidimensionais. O resultado da aplicação desta abordagem no exemplo da Figura 1.1(a) resultará em três registradores sendo alocados, respectivamente, para as referências $a[i], a[i+1], a[i-1]$. No entanto, se a arquitetura alvo para a qual este trecho de programa estiver sendo gerado possuir somente um registrador de endereçamento, dois destes registradores serão *spilled* (transbordados) na memória. Como consequência disto, os ganhos referentes ao uso das operações de auto incremento/decremento são quase que anuladas devido às instruções adicionais de LOAD e STORE necessárias para restaurar e salvar o valor do registrador de endereçamento.

Com o objetivo de contornar este problema, reduzindo o número de registradores *spilled*, este trabalho propõe também uma extensão do trabalho de Araujo no sentido de permitir a alocação do mesmo registrador de endereçamento para referências que ocorrem em diferentes blocos básicos (Capítulo 3). Para isto, *Static Single Assign Form (SSA)* [12] será redefinida para o problema em questão, tornando possível o uso de um mesmo registrador entre referências que ocorrem em vários blocos básicos do programa. Dentro desta estratégia, sempre que a cobertura por caminhos do IG requerer um número de registradores de endereçamento maior do que o presente na arquitetura alvo, sucessivas operações de junção de caminhos serão aplicadas na tentativa de se reduzir o número de registradores alocados. Para efetuar a junção de dois caminhos, instruções adicionais de redirecionamento poderão ser necessárias, quando não for possível usar auto-incremento(decremento).

Este trabalho mostra ainda que uma variação *greedy*(gulosa) deste algoritmo pode resultar, em muitos casos, em soluções tão boas quanto aquela resultante de uma solução exata baseada na cobertura por caminhos. O exemplo da Figura 1.1(b) mostra o resultado da aplicação deste algoritmo no trecho de programa da figura Figura 1.1(a), para o caso em que a arquitetura alvo tem somente um registrador de endereçamento. Como resultado apenas uma instrução de redirecionamento foi utilizada (`p += -2`) ao invés de várias instruções de LOAD/STORE necessárias a realização do *spill* de dois registradores.

As técnicas propostas neste trabalho podem ser utilizadas em qualquer arquitetura que tenha modo de endereçamento indireto, incluindo a maioria dos processadores CISCs (ex. Motorola 68000) e todos os DSPs. A validação das técnicas foi feita usando um compilador de produção e trechos de programas da empresa Conexant Systems Inc. (California, EUA), que foram cedidos para a realização de pesquisas acadêmicas em compiladores.

1.4 Organização da dissertação

Este texto encontra-se organizado da seguinte forma.

- No Capítulo 2 são introduzidos conceitos importantes, tal como distância e grafo de indexação, que serão utilizados para a formulação do problema de alocação de registradores de endereçamento como um problema de cobertura em grafos. Este capítulo generaliza o conceito de distância de indexação proposto por Araujo et al [3], para o caso de referências a elementos de arrays multidimensionais em laços aninhados. Neste capítulo é formulado também o problema de alocação de registradores de endereçamento (AIA) como um problema de cobertura em grafos usando caminhos e ciclos, para o caso de referências que ocorrem dentro de blocos básicos.

- O Capítulo 3 mostra como se resolver o problema de alocação de AR's para referências que ocorrerem espalhadas em comandos condicionais distintos (ex. corpo de sentenças *if-then-else*. Estes casos, que são frequentes em programas reais, merecem um tratamento especial. Neste capítulo, uma variação de *Static Single Assign Form* é usada para resolver este problema. Neste capítulo também é apresentada uma heurística que objetiva reduzir o número de registradores sempre que a solução apresentada no capítulo 2 requerer mais registradores de endereçamento que o número de registradores presentes na arquitetura. Por fim, é apresentada uma variação deste algoritmo usando o método guloso(*greedy*).

- O Capítulo 4 mostra a análise dos resultados obtidos após a implementação das técnicas dos capítulos 2, 3 e 4 em um compilador comercial pertencente à Conexant

System Inc.

- O Capítulo 5 conclui este trabalho e apresenta os possíveis trabalhos futuros que podem ser realizados como extensões ao aqui apresentado.

Capítulo 2

Alocação Local Usando o Grafo de Indexação

Science may never come up with a better office-communication system than the coffee break.

Earl Wilson

Este capítulo apresenta inicialmente, na Seção 2.1, uma revisão do conceito de representação de um programa sob a forma de um grafo dirigido. Em seguida, a Secção 2.2 são estabelecidas as condições que as referências a elementos de arrays em um laço devem satisfazer para que possam ser utilizadas nas técnicas de alocação propostas neste trabalho. Na Seção 2.3 é introduzido o conceito de distância de indexação para o caso unidimensional. A Seção 2.4 generaliza o conceito de distância de indexação para o caso multidimensional. Uma vez definidos os conceitos de distância, o Grafo de Indexação (IG)[1] é definido na Seção 2.5 como um mecanismo para resolver o problema de alocação de registradores de endereçamento para as referências aos elementos de um array no interior de laços. O problema de Alocação de Índice de Array é mostrado na Seção 2.6. A Seção 2.7 apresenta uma solução para o problema de alocação de registradores de endereçamento baseada na cobertura por caminhos do Grafo de Indexação (IG). A solução apresentada neste capítulo objetiva resolver apenas o caso simples do problema de alocação de registradores de endereçamento, no qual o laço contenha somente um bloco básico. No próximo capítulo será apresentada a generalização desta técnica para os casos em que referências ocorrem no interior de comandos condicionais.

[1] Do original em inglês: *Indexing Graph*

2.1 Grafo de Fluxo de Controle

As sentenças de um programa são organizadas sob a forma de blocos básicos. Um bloco básico é uma sequência de sentenças na qual o fluxo de execução do programa entra pela primeira sentença e somente pode sair pela última, sem a possibilidade de parada ou desvio. Um *Grafo de Fluxo de Controle* (CFG[2]) é um grafo dirigido cujos vértices são blocos básicos de um procedimento, incluindo dois vértices: `Entrada` e `Saída`. Existe uma aresta de `Entrada` para qualquer bloco básico no qual o procedimento pode entrar, e existe uma aresta para `Saída` de todo bloco no qual o procedimento pode ser terminado. As outras arestas no CFG representam as transferências de controle entre os blocos básicos. Suponha que cada vértice esteja num caminho do vértice `Entrada` para o vértice `Saída`. Para cada vértice X, um *predecessor* de X é qualquer vértice Y com uma aresta $Y \rightarrow X$ no CFG. O conjunto *Pred(X)* é o conjunto de todos os predecessores de X. De maneira análoga, um *sucessor* de X é qualquer vértice Y com uma aresta $X \rightarrow Y$ no CFG. O conjunto *Succ(X)* é o conjunto de todos os sucessores de X . Um vértice com mais de um sucessor é um *vértice de desvio*. Um vértice com mais de um predecessor é um *vértice de junção*.

Os conceitos de *dominância* e *pós-dominância* serão utilizados no decorrer deste trabalho. Um vértice *d* domina o vértice n, escrito como *d dom n*, se todo caminho de execução do vértice `Entrada` para o vértice *n* passa obrigatoriamente pelo vértice *d*. De maneira análoga é definido o conceito de pós-dominância. Um vértice *d* pós-domina um vértice *n*, escrito como *d pdom n*, se todo caminho de execução partindo de *n* para o vértice `Saída` passa obrigatoriamente por *d*.

2.2 Referência a Elementos de Arrays em Laços

O primeiro passo no processo de alocação de registradores de endereçamento é identificar as referências a elementos de arrays que ocorrem no interior de um laço. Para isto, uma série de restrições devem ser satisfeitas. Estas restrições determinam quais as referências são válidas para a aplicação das técnicas de alocação de registradores de endereçamento deste trabalho.

Suponha que uma sequência de referências a elementos de array, como as mostradas na Figura 2.1(a), ocorram no interior de um laço. Considere também que as referências ocorram em um único bloco básico. Esta restrição não é um requerimento das técnicas que serão apresentadas, como será visto no Capítulo 3. Suponha também que a variável

[2]Do original em inglês: *Control-Flow Graph*

de indução i seja modificada linearmente por um valor inteiro, e que os índices dos arrays dentro do laço sejam funções lineares de i, ex. $f(i) = a * i + b$, onde a e b são inteiros. Estas suposições não são sérias restrições, já que a maioria das referências a elementos de arrays que ocorrem no interior de laços as satisfazem. No método de geração de código adotado aqui, as partes da árvore sintática de expressões correspondentes às referências a elementos de arrays não são decompostas em suas operações atômicas, ou seja, as referências são mantidas até o final do escalonamento das intruções no programa. Suponha também que a *Eliminação de Subexpressões Comuns* [1] não seja realizada para o caso de referências a elementos de arrays, e que *Eliminação de Variável de Indução* é usada para otimizar o laço. As técnicas de alocação deste trabalho consistem em identificar as referências a elementos de arrays no interior de um laço e alocar o menor número possível de registradores de endereçamento para realizar estas referências. Para isto, as técnicas farão uso da propriedade de auto-incremento(decremento) de modo a verificar se é possível alocar o mesmo registrador de endereçamento para mais de uma referência.

Considere, por exemplo, uma referência na forma `vector[a*i+b]`, na qual `i` é a variável de indução, `a` e `b` são inteiros. Esta referência satisfaz todas as condições citadas anteriormente. Suponha que esta referência ocorra no interior de um laço em que a variável de indução `i` tenha o valor inicial i_0. O cálculo do endereço do primeiro elemento a ser referenciado, isto é `&vector[`i_0`] + k`, é atribuído a um `ar` fora do laço associado a variável de indução `i`, e a referência a `vector[a*i+k]` será feita através de `ar`. Esta estratégia é tradicionalmente utilizada para compilar referências a arrays em laços [29]. Ela requer que otimizações como eliminação de variáveis de indução e remoção de código invariante [1, 29] estejam presentes no compilador. As técnicas que serão mostradas neste trabalho melhoram esta abordagem, permitindo que um mesmo `ar` possa ser atribuído para mais de uma referência no interior do laço e também entre referências em iterações consecutivas do laço.

Outro ponto importante para fins da alocação de registradores de endereçamento é a ordem em que as referências ocorrem após o escalonamento. Esta ordem determina a sequência em que as referências serão realizadas no código gerado. A Figura 2.1(b) mostra a sequência de referências após o escalonamento das instruções do laço. O valor entre parênteses indica a ordem de escalonamento de uma referência.

No decorrer deste trabalho a alocação de registradores de endereçamento será realizada somente para referências que satisfazem as condições mencionadas acima. O algoritmo mostrado na figura Figura 2.2 será utilizado para este fim. A função `find_array_references` recebe como argumentos o conjunto `loop_nodes` de vértices do CFG correspondentes ao laço, e a lista `list_nodes`, na qual são armazenadas as referências encontradas no laço. Em `list_nodes`, as referências estão ordenadas de acordo com a ordem de escalonamento

```
for (i = 2; i < N + 2; i++)
{
  a [ i ]   = a [i - 2] + a [i + 1] - a[i - 1];
  a [i - 1] = 2 * a [i + 2];
}
```

(a)

```
for (i = 2; i < N + 2; i++)
{
  a [i - 2]      (1)
  a [i + 1]      (2)
  a [i - 1]      (3)
  a [ i ]        (4)
  a [i + 2]      (5)
  a [i - 1]      (6)
}
```

(b)

Figura 2.1: (a) Um laço típico de um programa dedicado. (b) Sequência de referências a um array após o escalonamento das instruções do laço.

em que ocorrem no programa. Este algoritmo será utilizado pelas duas técnicas a serem apresentadas nos capítulos seguintes.

2.3 Distância de Indexação

Para se implementar endereçamento indireto, as arquiteturas possuem unidades de *hardware*, conhecidas como unidades de geração de endereços (AGUs[3]), que realizam o cálculo de endereços rapidamente. Uma AGU possui um conjunto de registradores de endereçamento (ars[4]) e uma Unidade Lógica Aritmética (ALU[5]) que realiza operações tais como incremento/decremento. Os ars são usados para apontar para uma posição de memória onde o dado desejado está armazenado. Outras arquiteturas possuem AGUs mais elaboradas, que permitem que se adicione ou subtraia um valor imediato (*deslocamento*) a um ar. Para o caso de processadores dedicados, os deslocamentos são usualmente armazenados em registradores, e não codificados nas instruções, de forma a economizar bits para a codificação das instruções. Isto permite a codificação das instruções em um formato menor. Esta é a forma de endereçamento encontrada em muitos programas dedicados.

A possibilidade de um programa usar os modos de endereçamento presentes na AGU é medida pela *Distância de Indexação*. A distância de indexação mede a distância em memória entre duas referências a um array dentro do laço. Suponha que a AGU alvo

[3]Do original em inglês: *Address Generation Unit*
[4]Do original em inglês: *Address Registers*
[5]Do original em inglês: *Arithmetic and Logic Unit*

```
get_references(node_ptr node, list list_nodes){

   if(node->class == ARRAY_REFERENCE){

        if(node->reference satisfies vector[a*i+k])
             insert_node(node, list_nodes);
        else
             return;
    }
    else
     for each son_{i} of node
           get_references(son_{i},list_nodes);
}

find_array_references(node_set loop_nodes, list list_nodes)
{
    for each block in loop_nodes do
      for each command_node in block do
          get_references(command_node,list_nodes);
}
```

Figura 2.2: Pseudo-código do algoritmo para identificação das referências no interior de um laço.

possua modo de endereçamento de auto-incremento(decremento). Uma referência a um array será mapeada, a partir de agora, em um *acesso*, como definido abaixo.

Definição 2.3.1 *Seja acesso*$(r) = n$ *a função que mapeia uma referência de um elemento de um array* `r` *em um inteiro* n *(* $n = 1,2,...$*), onde* n *é a ordem em que* r *aparece após o escalonamento das instruções no interior de um laço. Nós dizemos que* n *é o acesso ao elemento do array referenciado por* `r`.

Seja $a[i]$ uma referência a um elemento de array. Considere, de agora em diante, que o tamanho do tipo básico do elemento $a[i]$ é igual a uma posição de memória.

Definição 2.3.2 *Sejam* a_1 *e* a_2 *duas referências, e* B_1 *e* B_2 *seus correspondentes blocos básicos. A referência* a_1 *domina a referência* a_2 *se e somente se* B_1 *dom* B_2*. De maneira análoga,* a_1 *pós-domina a referência* a_2 *se e somente se* B_1 *pdom* B_2.

Os conceitos de dominância e pós-dominância entre referências a elementos de arrays

são simples extensões dos conceitos similares para blocos básicos [1]. Suponha também que o conceito de dominador imediato e pós-dominador imediato [29] é o mesmo que aquele para blocos básicos.

Definição 2.3.3 *Sejam n_1 e n_2 acessos a um array. O acesso n_1 é dito ser menor(maior) que o acesso n_2 , denotado por $n_1 < n_2$ ($n_1 > n_2$), se e somente se n_1 precede (sucede) n_2 na ordem de escalonamento em que ocorrem no programa.*

Note que uma consequência imediata da definição acima é que todos os algoritmos e técnicas propostas neste trabalho somente ocorrem após o escalonamento das referências no programa ter ocorrido.

Toda referência a array na forma `vector[a*i+b]` pode ser representada por meio de uma *tripla* $t = (a, i, b)$, na qual `i` representa a variável de indução, `a` e `b` inteiros. O valor `a` é dito ser o primeiro coeficiente da tripla e `b` o último.

Definição 2.3.4 *A diferença entre duas triplas $x = (a, i, b)$ e $y = (c, i, d)$, representada por $x - y$, é a tripla $z = (a - c, i, b - d)$. A operação de diferença só é definida para triplas que possuem a mesma variável de indução* `i`.

Exemplo 1 A Figura 2.1(b) representa o código resultante após o escalonamento das instruções do laço da Figura 2.1(a). Para simplificar a análise, apenas as referências aos elementos do array `a` são mostradas em (b). Um número (em parênteses) indica o acesso de cada referência, correspondendo a ordem em que a mesma ocorre no escalonamento final. Por exemplo, acesso(`a[i-1]`) = 6.

A meta aqui é alocar um `ar` para cada referência a array no laço. Além disso, procurar-se-á minimizar o número de `ar's` necessários para endereçar todos as referências no interior do laço, maximizando assim o número de referências que podem compartilhar o mesmo `ar`. Para identificar se é possível compartilhar o mesmo `ar` entre duas referências, o conceito de *distância de indexação* é introduzido a seguir.

Definição 2.3.5 *Sejam $n_1 = (a, i, b)$ e $n_2 = (a, i, d)$ triplas representando referências ao mesmo array, e passo, o incremento do laço que as contém. A distância de indexação entre as triplas n_1 e n_2 é o valor inteiro:*

$$d(n_1, n_2) = \begin{cases} |d - b| & se\ n_1 < n_2 \\ |d - b + a * passo| & se\ n_1 > n_2. \end{cases}$$

Observe na Definição 2.3.5 que a distância de indexação só é definida para as triplas que possuem o mesmo coeficiente `a` e a mesma variável de indução `i`. De outro modo, $d(n_1, n_2)$ resultará em uma tripla e não em um inteiro positivo.

Exemplo 2 Considere por exemplo as referências do array a na Figura 2.1(b), em que *passo* = 1. A distância de indexação $d(1,4) = |0 - (-2)| = 2$. Uma vez que a distância de indexação é maior do que um, nenhuma operação de auto-incremento(decremento) pode ser usada para atualizar o registrador de endereçamento que aponta para o acesso (1), de forma que passe a apontar para o dado requerido pelo acesso (4). Por outro lado, desde que $d(4,1) = 1$, uma operação de auto-incremento(decremento) pode ser usada para redirecionar o registrador de endereçamento associado ao acesso (4) tal que ele aponte para o acesso (1). Note que isto ocorrerá quando o acesso (1) for atingido a partir do acesso (4), na próxima iteração do laço.

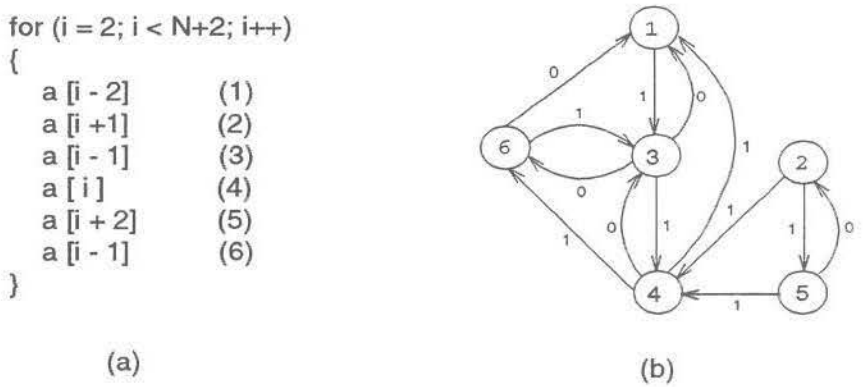


Figura 2.3: (a) Laço típico em programas dedicados. (b) IG correspondente.

2.4 O Caso Multidimensional

A Definição 2.3.5 apresenta o conceito de distância de indexação levando em consideração somente arrays unidimensionais, ou seja, somente vetores. As técnicas que serão apresentadas neste trabalho, para o problema de alocação de registradores de endereçamento, são independentes das dimensões do array.

Tal como para o caso de vetores, as mesmas restrições impostas aos índices de uma referência (descritas na Seção 2.2) devem ser satisfeitas para o caso multidimensional. Seja $a[i_1]...[i_n]$ uma referência multidimensional e j a variável de indução do laço de um índice i_j, e $1 \leq j \leq n$. A referência a é considerada válida se as variáveis de indução de cada índice i_j de a forem linearmente alteradas em cada iteração do laço . Além disto, cada índice i_j deve ser uma função linear de j, ou seja, $i_j = f(j) = a * j + b$.

Seja $r[i_1]...[i_n]$ uma referência n-dimensional de um array r, sendo n o número de

dimensões de r. Cada índice i_j, $1 \leq j \leq n$, de r pode ser representada na forma de uma tripla $t = (a, j, b)$, sendo j a variável de indução associada ao índice i_j, a e b inteiros. A referência r pode ser então representada pela seqüência de n-triplas:

$\mathtt{t}_n = \{(a_1, j_1, b_1), ..., (a_n, j_n, b_n)\}$.

Suponha que o tamanho do tipo básico de um elemento de array multidimensional seja uma posição de memória, tal como no caso unidimensional, e que, o mapeamento dos elementos do array na memória é do tipo *row major* [1], ou seja, o armazenamento se dá das dimensões menores para as maiores. Considere também uma referência $r[i_1]...[i_n]$ cujo laço mais interior que a contém possui variável de indução j. Além disto, suponha que j afeta a dimensão k de r. Ou seja, existe uma tripla $i_k = (a_k, j, b_k)$. Note que, se $k = 1$ então as iterações consecutivas de j tem como resultado o acesso contínuo dos elementos de r na memória. Neste caso, o modo auto-incremento pode ser usado para acessar r durante iterações consecutivas do laço. Por outro lado, se $k \neq 1$, então iterações consecutivas do laço controlado por j irão resultar em saltos de posições de memória cujo tamanho depende do tamanho de todas as dimensões de r menores que k. Neste caso, o salto dado por r entre iterações do laço é igual ao produtório do tamanho das dimensões menores que k. Deste modo, podemos definir o *deslocamento dimensional* $D(k)$ para cada uma das dimensões de um array como:

$$D(k) = \begin{cases} 1 & \text{se } k = 1. \\ \prod_{j=k+1}^{n} tamanho(j) & \text{caso contrário.} \end{cases}$$

Usando o conceito de deslocamento dimensional podemos então redefinir a distância de indexação como:

Definição 2.4.1 *Sejam* $n_1 = \{(a_1, j_1, b_1), ..., (a_n, j_n, b_n)\}$ *e* $n_2 = \{(a_1, j_1, c_1), ..., (a_n, j_n, c_n)\}$ *triplas representando referências n-dimensionais de um mesmo array. A* distância de indexação *entre as triplas* n_1 *e* n_2 *é o valor inteiro:*

$$d(n_1, n_2) = \begin{cases} \sum_{k=1}^{n} |(c_k - b_k)| * D(k)) & se\ n_1 < n_2 \\ \sum_{k=1}^{n} |(c_k - b_k + a_k * passo)| * D(k)) & se\ n_1 > n_2. \end{cases}$$

Exemplo 3 Considere o algoritmo da Figura 2.4. Este algoritmo calcula o produto de duas matrizes a e b, resultando na matriz c. As matrizes são representadas pelos arrays a[N][N], b[N][N] e c[N][N], sendo N = 50. Todos os índices satisfazem as condições de linearidade requeridas. Considere as referências a[i][k] e b[k][j] na linha (7) do algoritmo no laço mais interior. Seja $acesso(a[i][k]) = 1$ e $acesso(b[k][j]) = 2$. Suponha

```
(1)     void multmat(){
(2)        int i,j,k;
(3)        for (i=0;i<N;i++){
(4)           for (j=0;j<N;j++){
(5)              c[i][j] = 0.0;
(6)              for (k=0;k<N;k++){
(7)                 c[i][j] += a[i][k]*b[k][j];
(8)              }
(9)           }
(10)       }
(11)    }
```

Figura 2.4: Algoritmo que realiza a multiplicação de matrizes.

que o registrador `ar1` tenha sido alocado para o acesso 1, e o registrador `ar2` para o acesso 2. Calculando-se a distância de indexação entre o acesso 1 e o mesmo acesso 1 na próxima iteração resulta: d(1,1) = 1, desde que o passo do laço é 1. De modo análogo, d(2,2) = $| 1 - 1 + 0 | * 1 + | 1 - 1 + 1 | * 50 = 50$. Desta forma, é possível utilizar o modo de auto-incremento na referência 1 de forma a redirecionar o registrador `ar1` para a próxima iteração do laço. Por outro lado, já que d(2,2) = 50, o modo de auto-incremento não poderá ser utilizado em `ar2`, e uma instrução de redirecionamento `ar2 += 50` deverá ser inserida no final do laço controlado por `k`.

A distância de indexação determinará a possibilidade do uso do mesmo registrador por mais de uma referência no laço. Ela é utilizada na definição do Grafo de Indexação, o qual é usado para se resolver o problema de alocação.

2.5 Grafo de Indexação

O Grafo de Indexação(IG[6]) procura identificar as oportunidades de se partilhar o endereçamento indireto e é utilizado para alocar um mesmo `ar` para uma ou mais referências no laço. O IG é definido como:

Definição 2.5.1 *Um* Grafo de Indexação *(IG) é um grafo dirigido em que cada vértice corresponde a uma referência a um elemento de array, e existe uma aresta* (n_1,n_2) *se e*

[6]Do original em inglês: *Indexing Graph*

somente se $d(n_1, n_2) \leq |k|$, *em que* k *é o modificador de pós-incremento disponível na* AGU $k = +1(-1)$ *para o modo de auto-incremento(decremento).*

Existe uma aresta (n_1,n_2) no IG sempre que operações da AGU puderem ser utilizadas para redirecionar o registrador de endereçamento associado à referência n_1, tal que ele passe a apontar para a referência n_2. O IG tem dois tipos de arestas. Uma aresta (n_1,n_2) é uma *aresta progressiva*[7] se $n_1 \leq n_2$, caso contrário será uma *aresta regressiva*[8]

Exemplo 4 O IG da Figura 2.3(b) foi construído a partir das referências ao array a no laço da Figura 2.3(a). O IG não é um grafo rotulado, os rótulos nas arestas na Figura 2.3 são usados apenas para ilustrar a distância de indexação entre o vértice de origem e de destino de cada aresta. Observe que arestas com distância de indexação unitária, como por exemplo (3,4), indicam a possibilidade de se utilizar o modo de auto-incremento(decremento) entre os acessos na ordem de escalonamento. Arestas regressivas, ex. (6,1), indicam a possibilidade de se utilizar operações de auto-incremento(decremento) através das iterações do laço.

O Algoritmo da Figura 2.5 é utilizado para construir um IG. A função que constrói o IG, `build_IG`, recebe como parâmetro de entrada uma lista de vértices, `list_nodes`, que formam um laço no CFG. A função aloca uma nova estrutura `IG` que será o grafo a ser retornado. Inicialmente, os elementos de `list_nodes` são inseridos no conjunto de vértices de `IG`. A função responsável pela inserção dos vértices é `insert_IG_node`. É importante ressaltar novamente que tanto em `list_nodes` quanto em `IG->nodes` os vértices estão ordenados de acordo com a ordem de escalonamento em que ocorrem no laço. Este ordenamento será importante para determinar a existência de uma aresta no IG. O próximo passo consiste em se determinar o conjunto de arestas do IG. A função `set_IG_edges` insere as arestas em um IG. Para isto, a função itera sobre o conjunto de vértices, verificando a possibilidade de inserção de uma aresta entre um vértice v e todos os vértices subseqüentes a v na ordem de escalonamento. Para isto é importante que a lista de vértices esteja ordenada. Uma vez determinadas as arestas, o grafo IG é então retornado.

2.6 Alocação de Índices de Array

O problema de Alocação de Índices de Array (AIA) consiste em alocar registradores de endereçamento para referências a elementos de um array dentro de laços, tal que o número

[7] Do original em inglês: *forward edge*
[8] Do original em inglês: *backward edge*

```
set_IG_edges(IGraph IG)
{
 list_node node, next_node;

  for( node = IG->nodes; node; node = node->next)
    for( next_node = node->next; next_node; next_node = next_node->next)

        if( node->base_array == next_node->base_array &&
            next_node pos_dominates node &&
            node       dominates     next_node &&
            |indexing_distance(node, next_node)| <= 1)
             insert_edge(IG, node, next_node);

}
insert_IG_node(IGraph IG, list_node node)
{
     IG->num_nodes++;
     if(!IG->nodes)
        IG->nodes = new_IG_node(node);

     else{
        IG->nodes->tail->next = new_IG_node(node);
        IG->nodes->tail = IG->nodes->tail->next;
     }
}
IGraph *build_IG( list list_nodes)
{
    IGraph *IG;

    // allocate IG and initialize
    IG = new_IG();

    // insert nodes into graph
    for each node in list_nodes
       insert_IG_node(IG, node);

    set_IG_edges(IG);
    return IG;
}
```

Figura 2.5: Pseudo-código do algoritmo para construção do Grafo de Indexação

total de registradores de endereçamento necessários seja minimizado. Para formalizar o problema de alocação de índices de array, o conceito de *caminho* e *ciclo* em um IG serão definidos.

Definição 2.6.1 *Um caminho $n_i \rightarrow n_j$ em um IG G é uma sequência de vértices distintos $(n_i, n_{i+1}, \ldots, n_j)$, , que representam referências a elementos de array, tal que (n_k, n_{k+1}) é uma aresta em G e $n_k < n_{k+1}$.*

Definição 2.6.2 *Um ciclo em um IG G é uma sequência de vértices na forma $(n_i, n_{i+1}, \ldots, n_j, n_i)$ tal que $(n_i, n_{i+1}, \ldots, n_j)$ forma um caminho em G, para $i, j > 0$.*

Um caminho em um IG indica a possibilidade de alocação de um mesmo ar para a sequência de referências que formam o caminho, indicando a possibilidade de se utilizar o modo de auto-incremento(decremento) entre as referências, sempre que necessário. Similarmente, um ciclo indica a possibilidade de um mesmo ar ser utilizado não somente entre referências na ordem do programa, mas também por referências na próxima iteração do laço.

Definição 2.6.3 *Uma cobertura $C = \{c_1, c_2, \ldots c_k\}$ de um IG é um conjunto de caminhos e/ou ciclos disjuntos em G, tal que para todo vértice $n \in G$, $n \in c_i$, c_i é um elemento de C. O valor $\mid C \mid = k$ é a cardinalidade de C.*

Dado que a cobertura é disjunta, para cada par de elementos c_1 e $c_2 \in C$, $c_1 \cap c_2 = \phi$. Note que a cobertura de um IG não é unica, ou seja, o mesmo IG poderá possuir duas coberturas com a mesma cardinalidade.

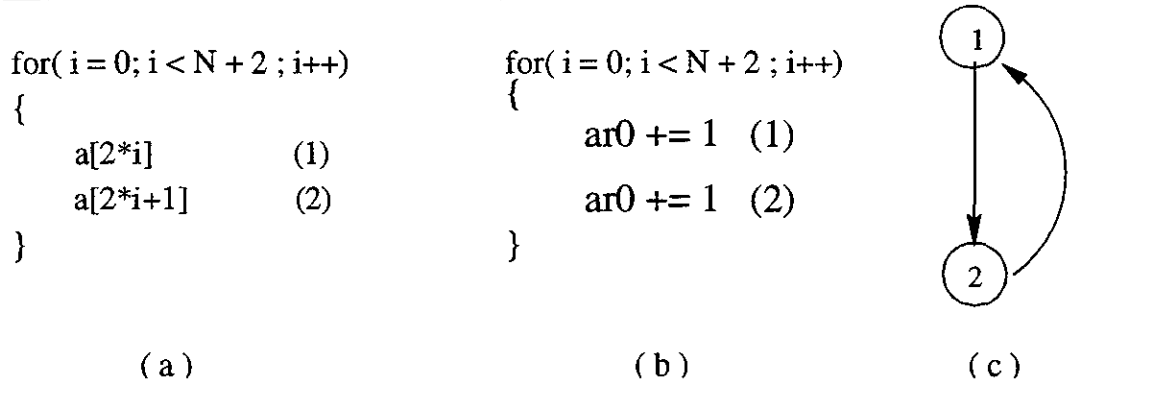


Figura 2.6: (a) Sequência de referências. (b) O registrador ar0 é incrementado após ambos os acessos, já que existe uma cobertura por ciclo (1,2,1). (c) Cobertura do IG.

Exemplo 5 A Figura 2.6(c) mostra o IG para os acessos da Figura 2.6(a) e a sua cobertura, para as quais temos as triplas $(1) = (2, i, 0)$ e $(2) = (2, i, 1)$. Calculando-se a distância de indexação entre as triplas (2) e (1) obtém-se: $d(2,1) = |(0 - 1 + (2 * 1)| = 1$. Isto indica a possibilidade de se utilizar auto-incremento do registrador ar0 através da iteração do laço, o que é mostrado na Figura 2.6(b).

2.7 Cobertura do IG

O problema de se minimizar o número de registradores de endereçamento para um dado IG pode ser formulado como o seguinte problema de otimização.

[IG Covering] *Dado um IG G, determinar a cobertura por caminhos/ciclos de G que minimiza o número total de caminhos e ciclos, e que tenha o menor número de caminhos. Suponha para o propósito deste problema que um vértice seja um caminho degenerado de tamanho zero.*

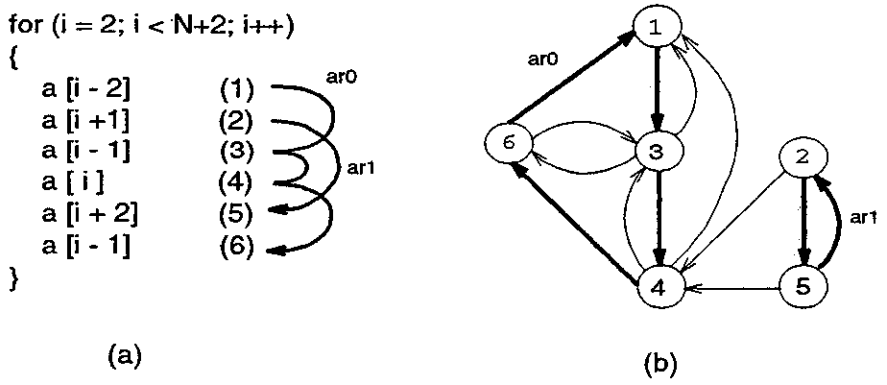


Figura 2.7: (a) Laço típico; (b) Cobertura do IG do laço em (a).

Cada caminho e ciclo no IG corresponde a um registrador de endereçamento. Além disso, ao contrário de um ar associado a um caminho, um ar associado a um ciclo permite que operações de auto-incremento(decremento) ocorram entre iterações consecutivas do laço. Note também que nem todos os ciclos são permitidos na definição acima. De acordo com a Definição 8, um ciclo em um IG pode percorrer somente uma aresta regressiva simples. A razão para isso é que os ciclos no IG devem permitir operações de auto-incremento(decremento) somente entre iterações consecutivas do laço, e não entre múltiplas iterações.

Exemplo 6 Considere o IG mostrado na Figura 2.7(b). Realizando-se a cobertura deste IG obtém-se dois ciclos mostrados em negrito. Cada ciclo corresponde a um registrador de endereçamento (ar_0 e ar_1). A alocação do registrador ar_1 para o ciclo (2,5,2) trata o caso de atualização do endereço da referência em (2) através da iteração do laço. O mesmo é verdadeiro para o registrador ar_0 e o ciclo(1,3,4,6,1).

A solução para a cobertura de um IG objetiva encontrar uma cobertura que tenha o menor número de caminhos (ou o o maior número de ciclos) de todas as coberturas que tenham a mesma cardinalidade (mínima). Para mostrar o porque, suponha que uma cobertura retorne um caminho p. Suponha que a distância de indexação entre o primeiro vértice do caminho (*head*) e o último (tail) seja maior do que um. Neste caso, não pode ser utilizado auto-incremento(decremento) para redirecionar o registrador de endereçamento para a primeira referência do caminho na próxima iteração do laço. Ao contrário, um ciclo já indica a possibilidade de se utilizar tal característica. Assim sendo, deseja-se uma cobertura que maximize o número de ciclos.

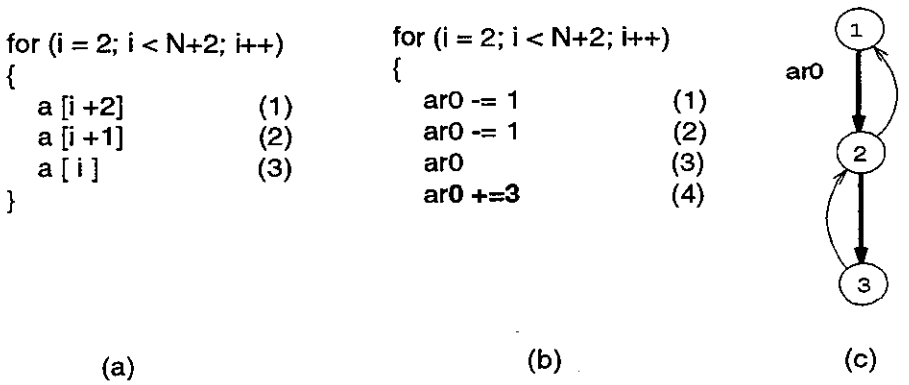


Figura 2.8: (a) Sequência original de referências a array; (b) É necessário adicionar 3 ao ar_0 para ajustá-lo para a primeira referência na próxima iteração; (c) A cobertura correspondente no IG.

Considere o exemplo da Figura 2.8(a), e sua cobertura na Figura 2.8(c). Desde que a AGU possui apenas o modo de auto-incremento(decremento), e $d(3,1) = |(i+2) - (i)+1| = 3 > 1$, então não existe nenhuma aresta de (3) para (1). A cobertura no IG retorna o caminho (1,2,3), ao qual é alocado o registrador ar_0. Desde que não existe aresta de (3) para (1), a instrução (4) é inserida no final do laço da Figura 2.8(b) para redirecionar o registrador de endereçamento ar_0, de tal forma que este aponte para o acesso (1) na próxima iteração. O desejável seria que a solução do problema de cobertura do IG resultasse na menor cardinalidade e no menor número de caminhos, de forma a

se evitar a introdução de instruções de redirecionamento de `ar's` no final do caminho. Infelizmente, encontrar uma solução exata para este problema é matematicamente difícil.

O problema de cobertura de um IG é similar ao problema de se encontrar a cobertura de ciclos disjuntos mínima de um grafo (MDCC[9]). O número de ciclos que cobrem todos os vértices de um grafo é conhecido como *índice de ciclo Hamiltoniano* [15]. Determinar o índice de ciclo hamiltoniano mínimo de um grafo foi mostrado ser NP-difícil . Ciclos em uma cobertura para o problema MDCC, ao contrário dos ciclos para a cobertura do IG, podem conter mais de uma aresta regressiva. Desta forma, concluimos que o problema de cobertura do IG possivelmente é NP-difícil, uma vez que é tão difícil quanto determinar o índice de ciclo Hamiltoniano mínimo. A prova direta disto é deixada como um problema em aberto para investigações futuras.

2.8 Cobertura Simples do IG

Dado que o problema de cobertura de um IG é NP-difícil, nós consideraremos algumas heurísticas para tratar este problema. A maneira mais simples é formular o problema tal que as operações de auto-incremento (decremento) entre iterações de um laço não sejam permitidas, como mostra a Figura 2.9. Com esta restrição, o IG torna-se acíclico e o problema original é reduzido ao problema de se determinar a cobertura por caminhos disjuntos mínima (MDPC[10]). Desta forma, podemos agora formular a versão simples do problema de cobertura do IG.

[Cobertura Simples do IG] *Dado um IG G, determinar a cobertura por caminhos disjuntos de G que minimize o número total de caminhos. Suponha, para o propósito deste problema, que um vértice é um caminho degenerado de tamanho zero.*

O problema MDPC foi estudado anteriormente por Boesch e Gimpel [5]. A solução que eles propuseram é baseada na solução de Hopcroft-Karp $O(n^{5/2})$ para o problema de emparelhamento em grafos bipartidos [21]. A idéia principal consiste em dividir cada vértice do grafo G em dois vértices n_1 e n_2, que pertencerão a conjuntos disjuntos. O vértice n_1 (n_2) é origem(destino) de todas arestas que saem de(chegam em) n. Fazendo isto, o grafo G' resultante é bipartido, com n_1 e n_2 pertencentes a conjuntos disjuntos. Aplicando-se o algoritmo de emparelhamento no grafo bipartido G' nenhum vértice em G' terá mais do que uma aresta saindo(entrando). Desta forma, o emparelhamento em G' forma uma cobertura em G que é composta somente por caminhos disjuntos. Suponha,

[9] Do original em inglês: *Minimum Disjoint Cycle Cover*
[10] Do original em inglês: *Minimum Path Disjoint Covering*

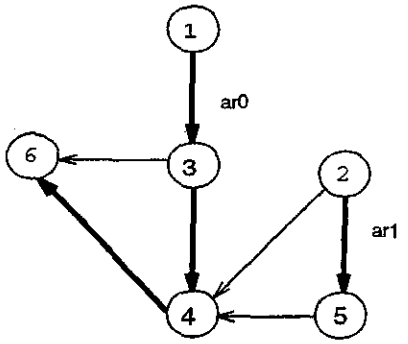


Figura 2.9: Resolvendo o problema do MDPC para um IG.

para isto, que um vértice é um caminho degenerado, e que todas as arestas tenham o mesmo peso durante o processo de determinação do emparelhamento mínimo em G'.

Exemplo 7 Solucionando o problema MDPC para o grafo acíclico da Figura 2.9 obtêm-se os caminhos (1,3,4,6) e (2,5), aos quais serão alocados os registradores $\mathtt{ar}_0$ e $\mathtt{ar}_1$. Ciclos ainda podem ser recuperados calculando-se a distância de indexação entre os vértices de início (*head*) e fim (*tail*) de cada caminho. Por exemplo, desde que $d(6,1) = 0 \leq 1$ e $d(5,2) = 0 \leq 1$, então nenhuma instrução de redirecionamento será necessária para redirecionar os apontadores $\mathtt{ar}_0$ e $\mathtt{ar}_1$, uma vez que no final de cada caminho, eles estarão apontando para a primeira referência de seu respectivo caminho. Neste caso particular, a heurística produziu o mesmo resultado obtido no Exemplo 6. No entanto, isto nem sempre ocorrerá, dado que a redução para MDPC é uma heurística para a solução de MDCC.

Capítulo 3

Alocação Global Usando SRF

Para que se pueda surgir lo possible,
es preciso intentar una y otra vez lo impossible

Provérbio

Este capítulo apresenta um método para se resolver o problema de alocação de registradores para referências que ocorrem no interior de comandos condicionais no corpo de um laço. Inicialmente, na Seção Seção 3.1, são apresentados os problemas referentes à alocação de um registrador para referências em comandos condicionais. Para tratar estes casos, o método apresentado utiliza uma variação do conceito de *Static Single Assignment Form* (SSA). Na forma SSA cada uso de uma variável possui apenas uma definição que o atinge. A Seção 3.2 apresenta a forma SSA e detalhes como converter o CFG para a mesma. Na Seção 3.3 a forma SSA é redefinida para tratar o caso de referências a elementos de arrays em laços. Esta nova representação é denominada *Forma de Referência Simples de Array (SRF*[1]*)*. Através do uso de SRF, a Seção 3.4 generaliza o método apresentado no capítulo anterior, baseado na cobertura por caminhos do IG, permitindo a alocação de registradores de endereçamento para referências no interior de condicionais. A Seção 3.5 apresenta uma heurística para reduzir o número de registradores de endereçamento sempre que a cobertura por caminhos do IG possuir uma cardinalidade maior do que o número de registradores presentes na arquitetura alvo. Esta heurística, baseada na junção de caminhos resultantes da cobertura do IG, utiliza a forma SRF para tornar eficiente o redirecionamento do registrador de endereçamento entre as referências. Na Seção 3.6 é apresentado o método guloso de alocação. A Seção 3.7 finaliza o capítulo mostrando o resultado da alocação, baseada em SRF, para parte de uma aplicação real pertencente à Conexant Systems Inc.

[1] *Do original em inglês:* Simple Reference Form

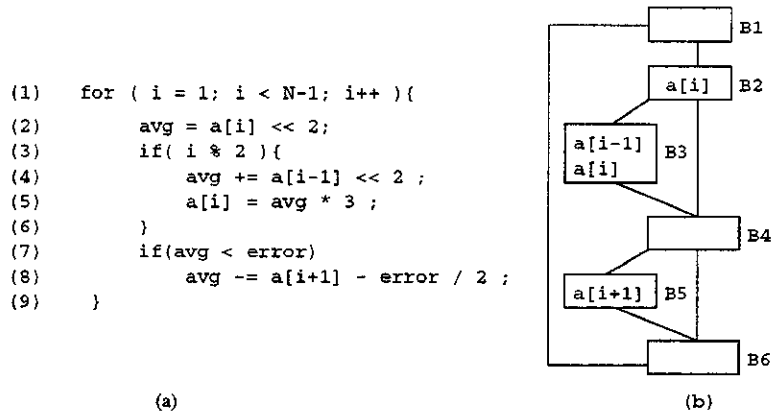


Figura 3.1: (a) Interior de um laço contendo comandos **if-then-else**. (b) A representação em CFG (Control Flow Graph).

3.1 Comandos Condicionais no Interior de Laços

Considere agora laços mais elaborados que contenham comandos *if-then-else* e/ou outros comandos condicionais aninhados. Por exemplo, o fragmento de programa da Figura 3.1(a). As referências `a[i]`, `a[i-1]` e `a[i+1]` pertencem a três blocos básicos diferentes: `B2`, `B3` e `B5`, como mostra o CFG da Figura 3.1(b). O bloco `B2` que contém a referência `a[i]` é sempre executado. Por outro lado, os blocos `B3` e `B5` , contendo, respectivamente, as referências `a[i-1]`, `a[i]` e `a[i+1]`, somente são executados se as correspondentes condições de teste forem satisfeitas.

Suponha que um registrador `ar` tenha sido alocado para duas referências quaisquer a_i e a_j que ocorrem, respectivamente, no interior de dois blocos básicos B_i e B_j. Suponha que os blocos B_i e B_j não sejam executados em toda iteração do laço. O registrador `ar` deve ser corretamente redirecionado sempre que o fluxo de execução atingir B_j sem passar pelo bloco B_i. O registrador `ar` também deve ser redirecionado para a próxima iteração do laço, ainda que o fluxo não passe por B_i ou B_j. Por exemplo, suponha que um registrador `ar` tenha sido alocado para a referência `a[i+1]` no bloco `B5` da Figura 3.1(b). O registrador `ar` é inicializado fora do laço e aponta para o elemento `&a[2]`, que é o elemento referenciado por `a[i+1]` na primeira iteração do laço. Desde que o bloco `B5` pode ou não ser executado, uma instrução de redirecionamento para a próxima iteração deve ser inserida em um bloco que seja executado a cada iteração do laço. Isto resulta na inserção da instrução de redirecionamento `ar += 1` no bloco `B6`, não sendo possível o uso do modo de auto-incremento(decremento), o que resulta no custo adicional de uma instrução.

Tomando-se novamente o exemplo da Figura 3.1(b), se o mesmo ar é alocado para as referências a[i] em B2 e a[i-1] em B3, instruções adicionais também devem ser inseridas de forma que o ar seja corretamente ajustado tanto entre essas referências como para a próxima iteração do laço.

Portanto, quando uma ou mais referências, para as quais se tenha alocado o mesmo registrador ar, ocorrem no interior de comandos condicionais, surge o problema de se determinar quais os caminhos de fluxo de execução irão requerer instrução de redirecionamento do registrador de endereçamento entre as referências. Ou seja, dada a ocorrência de uma referência a_i, deseja-se saber quais são as referências que atingem a_i por algum caminho de execução, bem como as referências que seguem a_i por algum caminho. Esta informação é usada para inserir instruções de redirecionamento. Observe que, não somente desejamos determinar quais são as referências que atingem um certo ponto do programa, como também precisamos garantir que, se um registrador ar é alocado para uma referência b, este registrador deve apontar para b no ponto do programa imediatamente anterior à sentença que b ocorre. Note que, se ar foi alocado a várias outras referências a_i que são executadas antes de b, então auto-incrememento (decremento) e/ou instruções de redirecionamento devem ser usadas para garantir que ar terá o mesmo valor antes de atingir b. Em outras palavras, desejamos usar uma representação para o CFG que garanta que qualquer referência seja atingida sempre pelo mesmo ar, contendo o mesmo valor. Uma representação para o CFG que garante esta propriedade é conhecida como *Static Single Assignment Form* [12].

3.2 A Forma SSA

Muitos problemas de análise de fluxo de dados precisam encontrar todos os usos de uma variável definida ou as definições de cada variável usada em uma expressão [2]. As cadeias DU (def-uso) e UD (uso-def) são estruturas de dados que tornam esta análise eficiente: para cada sentença no CFG, o compilador mantém uma lista de apontadores para todas as sentenças que usam a variável sendo definida na sentença (DU) e uma lista de apontadores para todas as definições das variáveis sendo usadas(UD).

Uma maneira mais eficiente de representar as cadeias DU e UD é obtida através de *Static Single Assignment Form* [13], denominada forma SSA, na qual cada variável possui somente uma definição no programa. Desta forma as cadeias DU e UD são simplificadas tornando vários algoritmos de otimização mais eficientes e simples de serem implementados, como por exemplo propagação de constantes, remoção de código invariante e *redução*

(1) a = x + y;	a_1 = x + y;
(2) b = a - 1;	b_1 = a_1 - 1;
(3) a = y + b;	a_2 = y + b_1;
(4) b = x * 4;	b_2 = x * 4;
(5) a = a + b;	a_3 = a_2 + b_2;
(a)	(b)

Figura 3.2: (a) Fragmento de código. (b) Programa em Static Single Assignment form.

de esforço[2], dentre outras [29].

O exemplo da Figura 3.2(a) mostra o fragmento de código de um bloco básico. Se somente blocos básicos são considerados, é fácil ver que cada instrução define um novo valor para uma variável e cada uso de uma variável pode ser modificado para usar a definição mais recente. No exemplo da Figura 3.2(b) cada variável foi modificada de forma a fazer uso de sua definição mais recente. Este código está na forma SSA, desde que cada variável possui somente uma definição. Isto foi obtido através da aplicação da técnica de *value numbering* [2].

No entanto, quando mais de um bloco básico é considerado, tal como na Figura 3.3, nem sempre o uso de uma variável possui apenas uma definição que o atinge. Por exemplo, na Figura 3.3(a), como a variável a recebe dois valores diferentes nos blocos `B1` e `B3`, não fica claro qual deles deveria ser usado no bloco `B4`.

Para solucionar este problema é introduzido o conceito de função-ϕ [29].

Definição 3.2.1 *função-ϕ é uma forma especial de atribuição que recebe como argumento um conjunto de definições $\{x_1, ..., x_n\}$ de uma variável x que atingem um ponto de junção no CFG (vértice no CFG com mais de um predecessor), e produz uma nova atribuição x_{n+1} para a variável x. Assim sendo, diz-se $x_{n+1} = \phi(x_1, ..., x_n)$.*

A Figura 3.3(b) mostra que os valores de a_1 (definido no bloco `B1`) e a_2 (definido no bloco `B3`) podem ser combinados usando a função $\phi(a_1,a_2)$. Todavia, ao contrário de funções matemáticas ordinárias, $\phi(a_1,a_2)$ produz a_1 se o fluxo de execução atinge o bloco `B4` pelo bloco `B2`, e produz a_2 se o fluxo vier pelo bloco `B3`.

A pergunta que surge então aqui é como a função-ϕ sabe qual o caminho de fluxo de execução tomado. Esta questão tem duas respostas que variam de acordo com o problema

[2] *Do original em inglês: strength reduction*

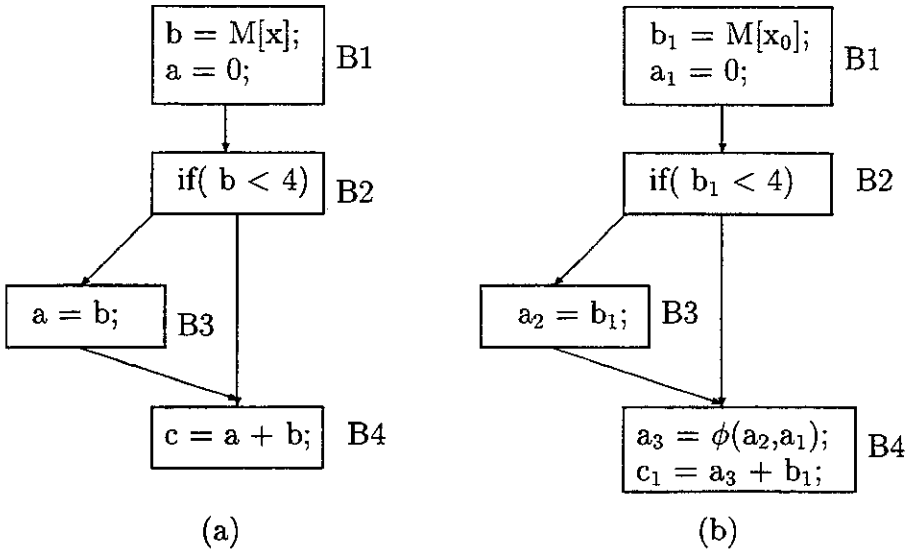


Figura 3.3: (a) Interior de um laço contendo comandos **if-then-else**. (b) A representação em CFG.

em questão.

- Se o programa precisa ser executado ou transformado para uma forma executável, as funções-ϕ podem ser implementadas inserindo-se instruções adequadas ao problema (ex. MOVE) em cada predecessor de um bloco que possui uma função-ϕ. Como será visto adiante, este é o caso que nos interessa já que as funções-ϕ irão determinar onde deverão ser inseridas instruções de redirecionamento dos registradores de endereçamento.

- Na maioria das otimizações deseja-se saber somente as conexões entre os usos e definições das variáveis, não sendo necessário desta forma executar uma função-ϕ específica. Nestes casos, pode-se ignorar qual o valor que será produzido pela função-ϕ, e utilizar a forma SSA apenas como um mecanismo para simplificar ou aumentar o escopo das otimizações.

3.2.1 Convertendo um Programa para Forma SSA

Durante o processo de conversão para forma SSA é preciso inicialmente determinar quais os pontos de junção do CFG nos quais devem ser inseridas funções-ϕ, para em seguida renomear todas as definições e usos de variáveis utilizando *índices*[3], como foi realizado nas figuras 3.2(b) e 3.3(b).

Critérios para inserção de funções-ϕ

Uma maneira óbvia de se converter um programa para forma SSA seria adicionar uma função-ϕ para toda variável a cada ponto de junção no CFG. No entanto, isto seria dispendioso e desnecessário. Por exemplo, o bloco B4 da Figura 3.3 não precisa de uma função-ϕ para b.

Deseja-se então encontrar o número mínimo de funções-ϕ tal que seja possível a conversão de um procedimento para a forma SSA. O CFG resultante é dito então estar em sua *Forma SSA Mínima*. Este problema é resolvido usando-se os conceitos de *Fronteira de Dominância*[4].

Fronteira de Dominância

Antes de apresentar o conceito de fronteira de dominância, extensões do conceito de dominância [29] serão apresentados a seguir:

Definição 3.2.2 *Um bloco x domina estritamente um bloco y (x sdom y) se x dom y e $x \neq y$.*

Definição 3.2.3 *Sejam a e b blocos básicos tal que $a \neq b$. Diz-se que a é o dominador imediato de b (a idom b) se a dom b e não existe nenhum outro vértice c tal que $c \neq a$ e $c \neq b$ para o qual a dom c e c dom b. Idom(b) denota o dominador imediato de b.*

Definição 3.2.4 *A fronteira de dominância (DF) de um vértice x é o conjunto de todos os vértices w tal que x domina um predecessor de w, mas x não domina estritamente w. Ou seja:*

$$DF(x) = \{\ y \mid \exists z \in Pred(y) \text{ tal que } x \text{ dom } z \text{ e } x \text{ !sdom } y\ \}$$

Suponha que ocorra uma definição de uma variável a em um vértice x do CFG. A fronteira de dominância do vértice x é formada por todos os vértices que são atingidos por uma

[3] *Do original em inglês: subscripts*
[4] *Do original em inglês: Dominance Frontier*

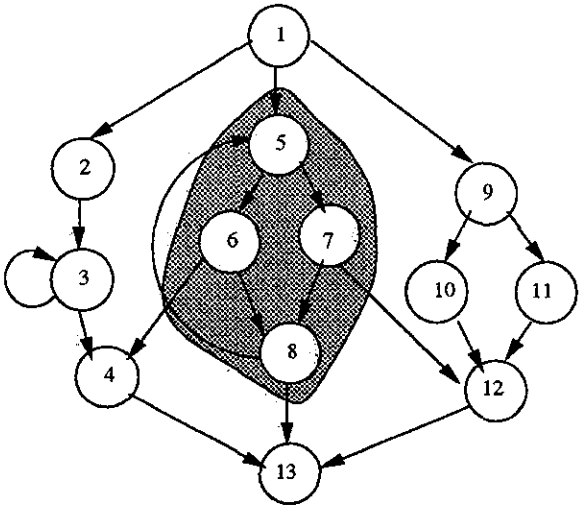


Figura 3.4: O vértice 5 domina todos os vértices da região cinza e sua fronteira de dominância inclui os vértices (4,5,12,13).

definição de a diferente da realizada em x. Ou seja, todos os vértices da fronteira de dominância requerem uma função-ϕ. Desta forma, os vértices delimitados pela fronteira de dominância são os vértices que são atingidos pela definição em x. A DF de um vértice x é unica, ou seja, o conjunto de vértices que formam a DF deve ser único.

Exemplo 8 A partir da definição acima, conclui-se que a fronteira de dominância de um vértice x é a fronteira entre vértices dominados e não dominados por x. Por exemplo a Figura 3.4 mostra que a fronteira de dominância do vértice 5 é composta pelos vértices (4,5,12,13). Do ponto de vista de análise de fluxo de dados, uma definição de uma variável a no vértice 5 atinge todos os vértices da área cinza, que são dominados por 5. A fronteira de dominância, por outro lado, indica os vértices que requerem funções-ϕ para a. Note que os vértices (6,7,8) são os vértices delimitados pela fronteira de dominância do vértice 5.

O algoritmo para a computação de *DF(x)* para todo x é quadrático no número de vértices do CFG. De um ponto de vista prático, em que modularidade implica em CFG com poucos vértices, o fato do cálculo de DF(x) ser $O(n^2)$ não constitui um problema de eficiência. No entanto, é possível se computar a DF em tempo linear [13]. Para que isto seja possível, o conjunto DF é dividido em duas componentes parciais $DF_{local}(x)$ e $DF_{up}(x)$, como:

$$DF_{local}(x) = \{y \in Succ(x) \mid idom(y) \neq x\}$$

$$DF_{up}(x, z) = \{y \in DF(z) \mid idom(z) = x \ \& \ idom(y) \neq x\}$$

Seja N o conjunto de vértices do CFG. A DF pode ser computada como:

$$DF(x) = DF_{local}(x) \cup \bigcup_{z \in N(idom(z)=x)} DF_{up}(x, z)$$

Para realizar o cálculo da fronteira de dominância, as equações acima são transformadas no pseudo-código mostrado no algoritmo da Figura 3.5. $Idom[x]$ é o conjunto de vértices que x domina imediatamente ($idom(x) = y$ se e somente se $x \in IDom[y]$). A função `calculate_IDom` é responsável pelo cálculo dos dominadores imediatos. `DF(x)` contém a fronteira de dominância de x. A função `post_order` retorna a lista dos vértices N do CFG em pós-ordem.

No exemplo 8, somente o vértice 5 possuía uma atribuição para uma variável a. No entanto, as variáveis de um programa geralmente são alvo de atribuição em mais de um bloco básico. Seja S o conjunto de vértices do CFG que atribuem para uma variável a. A fronteira de dominância de S é definida como:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

Desde que uma função-ϕ representa uma nova definição, é preciso inserir o resultado de $DF(S)$ no conjunto de definições S e recalcular $DF'((S \cup DF(S))$, repetindo-se isto sucessivamente até que nenhum outro vértice precise de uma função-ϕ. Desta forma a *Fronteira de Dominância Iterada*[5] $DF^+(S)$ é definida como

$$DF^+(S) = \lim_{i \to \infty} DF^i(S)$$

tal que $DF^1(S) = DF(S)$ e $DF^{i+1}(S) = DF(S \cup DF^i(S))$. Se S é o conjunto de vértices em que ocorrem atribuições para uma variável x, então $DF^+(S)$ é o conjunto de todos vértices que serão inseridas funções-ϕ para x.

[5] Do original em inglês: *Iterated Dominance Frontier*

```
list_nodes *computeDF(list_nodes N, node n ) {
    node y,z;
    sequence_of_node P;
    list_nodes *DF = new(vector(size(N)));
    // calculate the imediate dominators
    IDom = calculate_IDom(N);

    // P is the sequence of nodes in post-order
    P = post_order(N);

    for( i = 1; i < | P |; i++){
        DF[P[i]] = ∅;
        for each y ∈ Succ(P[i]) do
            if( y ∉ IDom(P[i]) then
                DF[P[i]] ∪= {y}

        for each z ∈ IDom[P[i]] do
            for each y ∈ DF[z] do
                if( y ∉ IDom(P[i]) then
                    DF[P[i]] ∪= y
    }
    return DF;
}
```

Figura 3.5: Pseudo-código do algoritmo que computa a fronteira de dominância.

O algoritmo da Figura 3.8 computa a fronteira de dominância iterada para um determinado conjunto S de vértices do CFG. O algoritmo considera que o cálculo da fronteira de dominância DF(x) tenha sido realizado para todos os vértices x do CFG. A função DF_Plus(S) possui complexidade quadrática do número de vértices do CFG no pior caso, mas é usualmente linear na prática [29].

Um exemplo da conversão de um programa para forma SSA pode ser visto no CFG da Figura 3.6. Usando o método iterativo da fronteira de dominância definido acima, serão computadas as fronteiras de dominância para as variáveis `k`, `i` e `j`. Observe que as variáveis `k` e `j` são definidas nos blocos `B1` e `B3`. Suponha, neste caso, que exista uma definição para `k` e `j` que atinja o vértice `Entrada`. Portanto S = { `Entrada`, `B1`, `B3`} para as variáveis `k` e `j`. Logo:

$DF^1(|\,\texttt{Entrada}, \texttt{B1}, \texttt{B3}\,|) = \{\texttt{B2}\}$

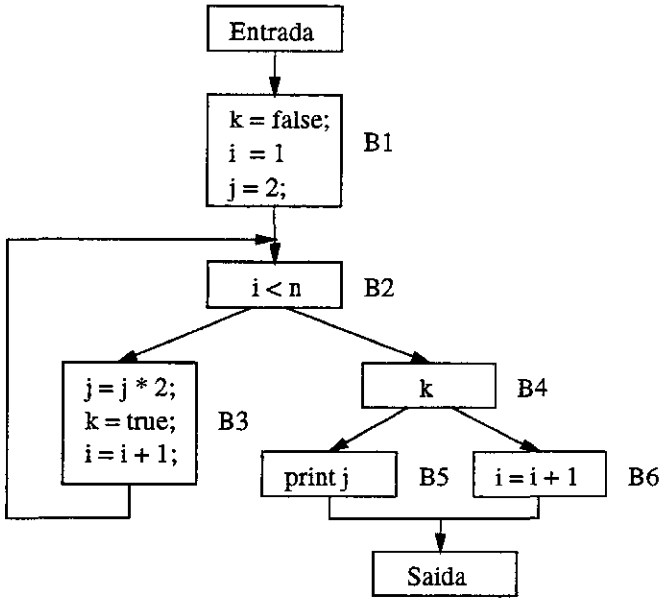


Figura 3.6: Trecho de programa a ser convertido para forma SSA.

$DF^2(|\ \texttt{Entrada}, \texttt{B1}, \texttt{B3}\ |) = DF(|\ \texttt{Entrada}, \texttt{B1}, \texttt{B2}, \texttt{B3}\ |) = \{\texttt{B2}\}$

para i temos S = {Entrada,B1,B3,B6}, logo:

$DF^1(|\ \texttt{Entrada}, \texttt{B1}, \texttt{B3}, \texttt{B6}\ |) = \{\texttt{B2}, \texttt{Saida}\}$

$DF^2(|\ \texttt{Entrada}, \texttt{B1}, \texttt{B3}, \texttt{B6}\ |) = DF(|\ \texttt{Entrada}, \texttt{B1}, \texttt{B2}, \texttt{B3}, \texttt{B6}, \texttt{Saida}\ |) = \{\texttt{B2}, \texttt{Saida}\}$

Desta forma, os blocos B2 e Saída são os blocos que necessitam de funções-ϕ. O bloco B2 tem uma função-ϕ para cada variável i, j e k e o vértice Saída possui uma para a variável i, como mostra a Figura 3.7. Note que se a variável i não estiver viva no bloco Saída não será necessário inserir uma função-ϕ para i neste bloco.

O último passo no processo de conversão de um programa para a forma SSA consiste em alterar todos os usos das variáveis, tal que cada uso refira-se a definição mais recente de uma variável, incluindo todas as definições adicionais representadas pelas funções-ϕ. Para o exemplo da Figura 3.6, obtém-se o programa na forma SSA ilustrado na Figura 3.7.

3.3 Forma de Referência Simples

Na Seção 3.2 foi introduzido o conceito da forma SSA. Foram apresentados também os métodos para a conversão de um programa para a forma SSA.

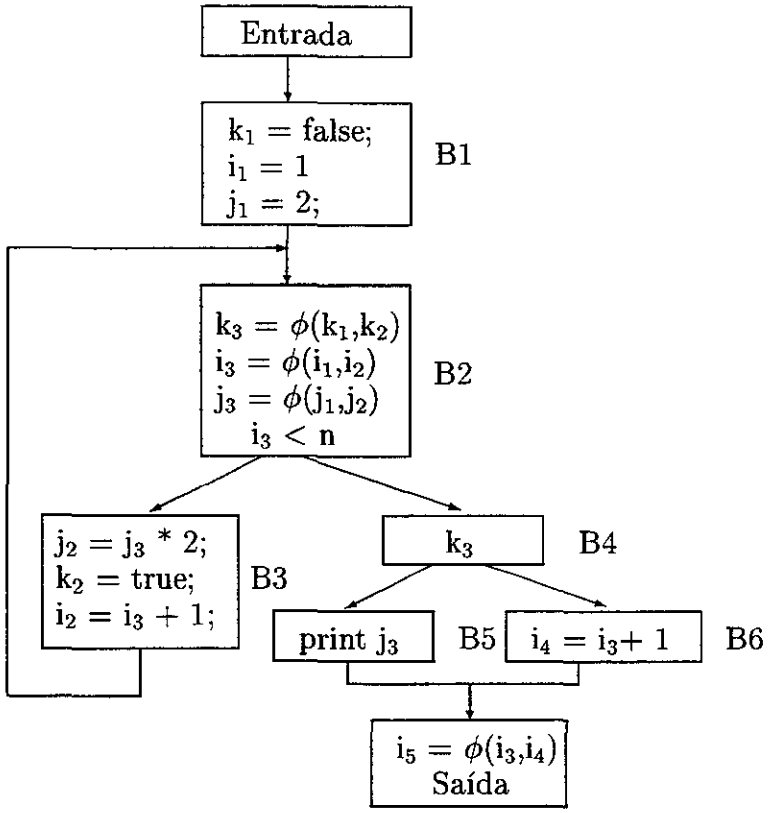


Figura 3.7: Resultado da conversão do grafo de fluxo da figura 3.6 para forma SSA.

A forma SSA foi apresentada com o intuito de resolver o problema de alocação de *ar's* para referências no interior de comandos condicionais. Para que seja possível a alocação de registradores de endereçamento para uma referência, independente de sua posição no CFG , o programa deve estar na *Forma de Referência Simples(SRF)* [6] que nós definimos como:

Definição 3.3.1 *Seja G o CFG de um programa R o conjunto de referências a elementos de um array. G é dito estar na Forma de Referência Simples (SRF) se cada referência a um array a_i pode ser atingida por uma única referência a_j.*

Note na definição 3.3.1 que o problema SRF é apenas um caso especial da forma SSA em que as variáveis são referências a elementos de arrays. Desta forma, os métodos e conceitos previamente definidos para transformar um programa na forma SSA são igualmente utilizados para a transformar um programa na forma SRF. A única diferença é que

[6] Do original em inglês: *Simple Reference Form*

```
list_nodes *DF_Plus(list_nodes N) {
     set_of_node D,DFP;
     boolean change = true;

     DFP = DF_Set(S);
     do{
          change = false;
          D = DF_Set(S ∪ DFP);
          if(D ≠ DFP){
               DFP = D;
               change = true;
          }
     }while(!changed)

     return DFP;
}
set_of_node DF_Set(set_of_node S){
     node x;
     set_of_node D = ∅;
     for each x ∈ S do
          D ∪= DF(x);
     return D;
}
```

Figura 3.8: Pseudo-código do algoritmo que computa a fronteira de dominância iterada.

em vez de variáveis, referências a elementos de array são consideradas.

Para se converter um programa para a forma SRF, tal como a conversão para a forma SSA, é necessário inicialmente determinar quais os vértices do CFG requerem funções-ϕ. As funções-ϕ tornam possível que o programa seja tranformado de forma a satisfazer as condições da Definição 3.3.1. Uma vez inseridas as funções-ϕ no CFG, o próximo passo é determinar quais as referências atingem uma função-ϕ, bem como as que são atingidas por uma função-ϕ, para isto será usada a *Análise de Referências.*

3.3.1 Análise de Referências

O item de dado utilizado durante a análise de referências é uma referência a um array. Seja R o conjunto de referências para o qual se deseja realizar a análise de referências.

Uma sentença s define uma referência $a \in R$ se a é usada em s. Uma sentença s mata uma referência $a \in R$ se outra referência $b \in R$ é usada em s e $a \neq b$. Para cada bloco básico, pode-se computar os conjuntos *r_gen[B]* como sendo o conjunto de referências geradas no bloco B e *r_kill[B]* como sendo o conjunto de referências mortas em B. A análise de referências é semelhante ao cálculo de *definições incidentes*[7] para o qual existem soluções eficientes bem conhecidas [1, 29]. Para o caso de CFGs redutíveis(estruturados) [1], a Análise de Fluxo de Dados Estruturada [8] pode ser aplicada.

Após a análise de referências ser realizada, cada sentença do programa possui um conjunto de referências que a atinge. Este conjunto é utilizado para computar, para cada sentença s, os seguintes conjuntos: (a) UD_s, a cadeia *uso-def* de referências que atingem s. (b) DU_s, a cadeia *def-uso* de referências que usam uma referência definida em s. Quando s é uma função-ϕ as referências em UD_s são usadas como argumentos da função-ϕ: $\phi(a_1, a_2, ..., a_n)$, $a_i \in UD_s$. Por exemplo, na Figura 3.9(a), $UD_\phi = \{a_1, ..., a_i, ..., a_n\}$ e $DU_\phi = \{b_1, ..., b_j, ..., b_m\}$.

3.4 Cobertura por Caminhos usando SRF

Na solução apresentada no capítulo anterior, o conjunto de vértices pertencentes ao IG representavam referências que estavam no mesmo bloco básico. No entanto, os laços de programas frequentemente possuem comandos condicionais, e o fato de uma referência `r` ocorrer em um comando condicional implica que a sentença contendo `r` pode ou não ser executada em toda iteração do laço.

Ao se converter o programa para a forma SRF, cada referência a_i é atingida por uma única referência a_j, o que torna possível a alocação de $ar's$ para referências em comandos condicionais. O próximo passo é decidir qual a referência resultante de uma função-ϕ. Esta referência é usada como uma referência intermediária entre todas as referências que chegam em uma função-ϕ e todas as que se seguem. Esta escolha influencia diretamente o desempenho do programa, pois dependendo da referência intermediária escolhida mais instruções de redirecionamento podem ser necessárias. Portanto, deseja-se escolher uma referência intermediária que resulte no menor número possível de instruções de redirecionamento, minimizando assim o impacto (custo) destas no desempenho do programa.

[7] Do original em inglês: *reaching definitions*
[8] Do original em inglês: *Structured Data Flow Analysis*

3.4.1 Custo das Instruções de Redirecionamento

Uma vez determinados os pontos do programa que devem receber funções-ϕ e os conjuntos UD_ϕ e DU_ϕ, torna-se necessário agora determinar as referências resultantes da aplicação das funções-ϕ. Deseja-se que a referência resultante seja tal que o custo das operações de redirecionamento entre as referências em DU_ϕ e UD_ϕ seja mínimo. Uma *referência real* é uma referência r_i pertencente ao conjunto $R = \{r_1, ..., r_n\}$ de referências que ocorrem no programa. Uma referência é dita ser *virtual* se ela for resultante de uma função-ϕ. Sejam a e b duas referências. Define-se $cost(a, b)$ como:

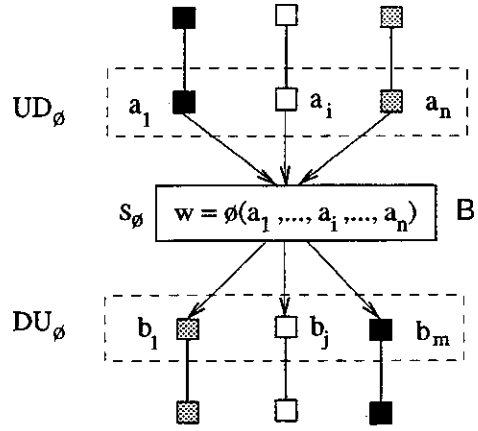
$$cost(a,b) = \begin{cases} 0, & se\ |d(a,b)| \leq 1 \text{ e } a \text{ é uma referência real, ou} \\ & se\ |d(a,b)| = 0 \text{ e } a \text{ é uma referência virtual,} \\ 1, & \text{caso contrário.} \end{cases} \quad (3.1)$$

A função custo retorna zero sempre que existir a possibilidade de uso de auto-incremento entre as referências. Note que quando a referência a é resultante de uma função-ϕ, cost(a,b) será zero somente se a referência $b = a$, $\forall b \in DU_\phi$.

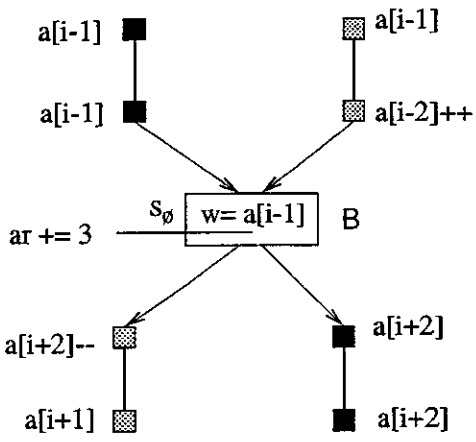
Seja ar o registrador de endereçamento atribuído para um conjunto de referências R de um array. Considere todas as instruções s_ϕ que tenham pelo menos uma referência de R como argumento, como por exemplo as mostradas na Figura 3.9. Note que as referências não pertencentes ao conjunto R são irrelevantes dado que se deseja realizar o redirecionamento somente para as referências em R, que dizem respeito ao mesmo array. Após ter-se realizado análise de referências, qualquer s_ϕ tem associada a ela os conjuntos UD_ϕ e DU_ϕ, com elementos $a_i \in UD_\phi$ e $b_j \in DU_\phi$. O valor de w depende de quão distantes estão os elementos em UD_ϕ e DU_ϕ . O objetivo aqui é selecionar a referência w que reduz o número de instruções de redirecionamento requeridas para o uso do mesmo registrador entre as referências em R. Isto pode ser realizado utilizando-se o seguinte algoritmo, ilustrado na Figura 3.9(b)-(e). Nestas figuras, O símbolo "++" ("- -") seguindo uma referência a um elemento de um array determina o modo de auto-incremento(decremento) para a referência.

Algoritmo 3.4.1 Algoritmo para determinar a referência resultante de uma função-ϕ.

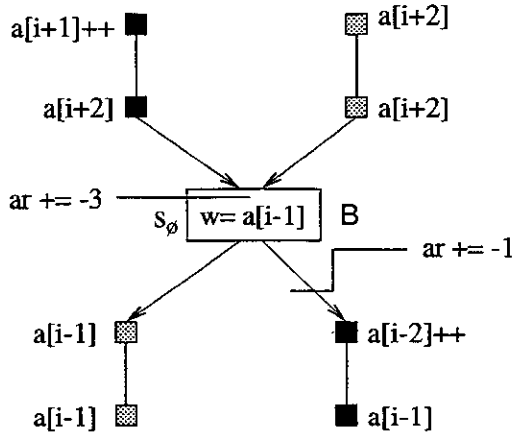
1. Se os elementos $b \in DU_\phi$ são os mesmos, mas os elementos de UD_ϕ são diferentes. Neste caso, w é a referência que minimiza $\sum_{i=1}^{|UD_\phi|} cost(a_i, w) + cost(w, b)$. A instrução s_ϕ é removida e instrução de redirecionamento $ar+ = d(w, b)$ é inserida em B, se $cost(w, b) = 1$. Inserir, na saída do bloco $P_i \in Pred(B)$ que contém a_i, uma



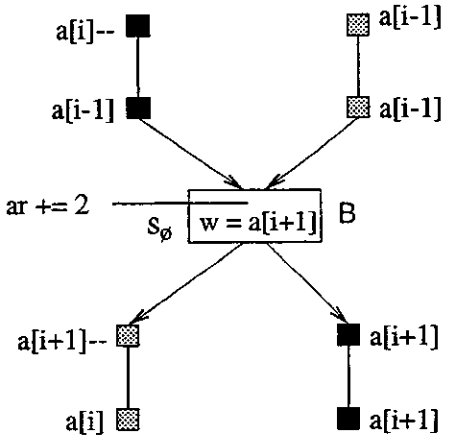
(a)



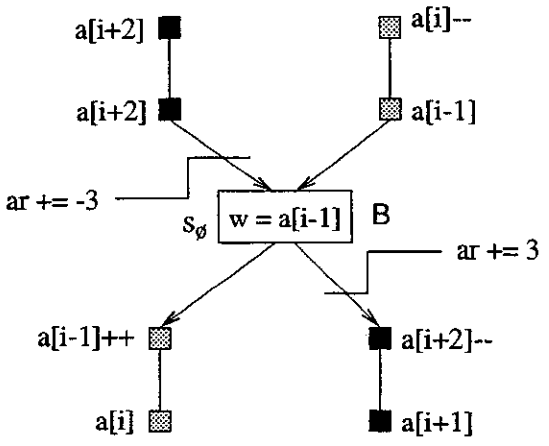
(b)



(c)



(d)



(e)

Figura 3.9: Exemplos de inserção de instruções de redirecionamento.

instrução de redirecionamento $ar+ = d(a_i, w)$, se $cost(a_i, w) = 1$; caso contrário, usar o modo de auto-incremento(decremento) adequado para a_i.

Exemplo 9 Considere, por exemplo, as referências da Figura 3.9(b). Note que $w = a[i-1]$ e todas as referências que são atingidas por s_ϕ são iguais, i.e., $b_j = a[i+2], \forall b_j \in DU_\phi$. Desde que $cost(a[i-1], a[i+2]) = 1$, a função-ϕ s_ϕ é substituída por uma instrução $ar+ = 3$ que redireciona ar de $w = a[i-1]$ para $a[i+2]$. Para a referência $a[i-2]$ é atribuído o modo de auto-incremento e desta forma ela é reescrita como $a[i-1]$ para os passos seguintes do algoritmo.

2. Se os elementos $a \in UD_\phi$ são os mesmos, mas os elementos de DU_ϕ são diferentes. Neste caso w é a referência que minimiza $cost(a, w) + \sum_{j=1}^{|DU_\phi|} cost(w, b_j)$. Remover s_ϕ e inserir dentro de B a instrução de redirecionamento $ar+ = d(a_i, w)$, se $cost(a_i, w) = 1$, caso contrário usar o modo adequado de auto-incremento(decremento) para a_i. Inserir, na entrada do bloco $S_j \in Succ(B)$ que contém b_j a instrução de redirecionamento $ar+ = d(w, b_j)$, se $cost(w, b_j) = 1$.

   Exemplo 10 Considere, por exemplo, as referências da Figura 3.9(c), para as quais tenha sido atribuído o mesmo registrador de endereçamento ar. Neste caso $w = a[i-1]$ e todas as referências que atingem s_ϕ são iguais, i.e. $a_i = a[i+2], \forall a_i \in UD_\phi$. Desde que $cost(a[i+2], a[i-1]) = 1$, a função-ϕ s_ϕ é substituída por uma instrução $ar+ = -3$ que redireciona o ar de $a[i+2]$ para $w = a[i-1]$. Além disso, desde que $cost(a[i-1], a[i-2]) = 1$, a instrução $ar+ = -1$ é inserida no caminho de $w = a[i-1]$ para $a[i-2]$.

3. Se os elementos de DU_ϕ (UD_ϕ) são os mesmos.
   Neste caso, w é a referência que minimiza $cost(a, w)+cost(w, b)$. Remover s_ϕ e inserir instruções de redirecionamento ou utilizar modo de auto-incremento(decremento) de acordo com os casos (1) e (2) acima.

   Exemplo 11 Considere, por exemplo, as referências que aparecem na Figura 3.9(d). Note que $w = a[i+1]$ e toda as referências em UD_ϕ (DU_ϕ) são as mesmas, i.e. $a[i-1]$ (a[i+1]). Desde que $cost(a[i+1], a[i+1]) = 0$, nenhuma instrução de redirecionamento é requerida de $w = a[i+1]$ para $a[i+1]$. Por outro lado, $cost(a[i-1], a[i+1]) = 1$ e desta forma uma instrução $ar+ = 2$ é requerida no bloco B.

4. Se os elementos em UD_ϕ (DU_ϕ) não são os mesmos.
   Neste caso, w é a referência que minimiza $\sum_{i=1}^{|UD_\phi|} cost(a_i, w) + \sum_{j=1}^{|DU_\phi|} cost(w, b_j)$. Remover s_ϕ e inserir, na saída do bloco $P_i \in Pred(B)$ que contém a_i, uma instrução de redirecionamento $ar+ = d(a_i, w)$, se $cost(a_i, w) = 1$; caso contrário, usar modo de

auto-incremento (decremento) para a_i. Inserir uma instrução de redirecionamento $ar+ = d(w, b_j)$ na entrada de $S_j \in Succ(B)$, se $cost(w, b_j) = 1$.

Exemplo 12 Considere, por exemplo, as referências na Figura 3.9(e). Neste caso $w = a[i-1]$, e os elementos dos conjuntos UD_ϕ e DU_ϕ são distintos dentro de cada conjunto. Desde que $cost(a[i+2], a[i-1]) = 1$, uma instrução de redirecionamento $ar+ = -3$ é inserida no final do bloco que contém a$a[i+2]$ para redirecionar ar de $a[i+2]$ para $w = a[i-1]$. De maneira similar, a instrução $ar+ = 3$ é inserida na entrada do bloco que contém $a[i+2]$.

3.4.2 Algoritmo para a Alocação Global

A técnica SRF permite que as referências que ocorrem no interior de um laço, independente do bloco no qual estejam, sejam representados por vértices no IG, tal como apresentado no Capítulo 2, desde que algumas restrições adicionais sejam satisfeitas para a existência de uma aresta no IG. A existência de uma aresta direcionada do vértice v_1 para v_2 implica na existência de um caminho no CFG que vai da primeira referência v_1 para a referência v_2, tal que, sempre que o fluxo de execução atinge o bloco contendo a referência representada por v_1, o fluxo também deve obrigatoriamente passar também pelo bloco contendo a referência representada pelo vértice v_2, na mesma iteração do laço. Ou seja, existe uma aresta entre duas referências v_1 e v_2 se as seguintes condições são satisfeitas:

1. O bloco que contém a referência do vértice v_1 dominar o bloco que contém a referência do vértice v_2.
2. O bloco que contém a referência do vértice v_2 pós-dominar o bloco que contém a referência do vértice v_1.

Desta forma, o algoritmo que cria as arestas no IG, mostrado na Figura 2.5 é alterado de forma a satisfazer as condições supra-citadas, resultando no pseudo código do algoritmo na Figura 3.10.

É importante ressaltar que se a cobertura do IG resultar em um caminho com um único vértice v_1, que representa uma referência no interior de um comando condicional, a técnica SRF deve ser aplicada para inserir as instruções de redirecionamento, desde que o fluxo pode tomar um caminho que não passa pelo bloco que contém a referência em v_1, em uma iteração do laço.

O algoritmo para alocação global de registradores de endereçamento consiste nos seguintes passos:

```
set_edges_IG(IGraph IG)
{
  list_node node, next_node;

   for( node = IG->nodes; node; node = node->next)
     for( next_node = node->next; next_node; next_node = next_node->next)

         if( node->base_array == next_node->base_array &&
             next_node pos_dominates node &&
             node       dominates     next_node &&
             |indexing_distance(node, next_node)| <= 1)
              insert_edge(IG, node, next_node);

}
```

Figura 3.10: Pseudo-código do algoritmo para determinar as arestas do IG.

Algoritmo 3.4.2 Algoritmo SRF

1. Identificar as referências no interior de um laço. O algoritmo para determinar todas as referências do interior de um laço é mostrado na Figura 2.2.
2. Construir o IG, inserindo os vértices e determinando as arestas.
3. Determinar a cobertura mínima C do IG.
4. Criar um novo bloco básico B na representação intermediária do programa. Este bloco será o *pre-header* do laço [1]. Desta forma, B terá como predecessores todos os predecessores do header do laço, e como sucessor o *header* do laço.
5. Realizar a análise de referências.
6. Para cada caminho $p_i \in C$, $0 \leq i <\mid C \mid$:
   (a) calcular as funções-ϕ necessárias para o redirecionamento do ar alocado. Todo caminho resultante da cobertura requer no mínimo uma função-ϕ , que é inserida no *header* do laço. Esta função-ϕ é devida a iteração do laço. Esta função-ϕ é denominada ϕ-header.
   (b) Determinar qual a referência o ar será inicializado. Esta informação pode ser obtida do conjunto DU_ϕ em ϕ-header. Note que as referências no conjunto UD_ϕ

de ϕ-header não podem ser consideradas, pois estas são referências devidas à iteração do laço.

(c) Inserir a inicialização do ar no pre-header do laço.

(d) Resolver as funções-ϕ no sentido contrário ao fluxo de execução do laço, ou seja, iniciando no último bloco do laço até o primeiro usando-se a aresta refluente do laço. Para se resolver as funções-ϕ utiliza-se o algoritmo 3.4.1.

(e) Modificar a representação intermediária da seguinte forma:

- Substituir as partes da representação intermediária que realizam as referências de array, representadas pelos vértices do caminho, para referências através do ar alocado para o caminho p_i.
- Determinar e inserir as instruções de redirecionamento necessárias para o correto redirecionamento do ar, fazendo uso do modo de auto-incremento(decremento) sempre que possível.

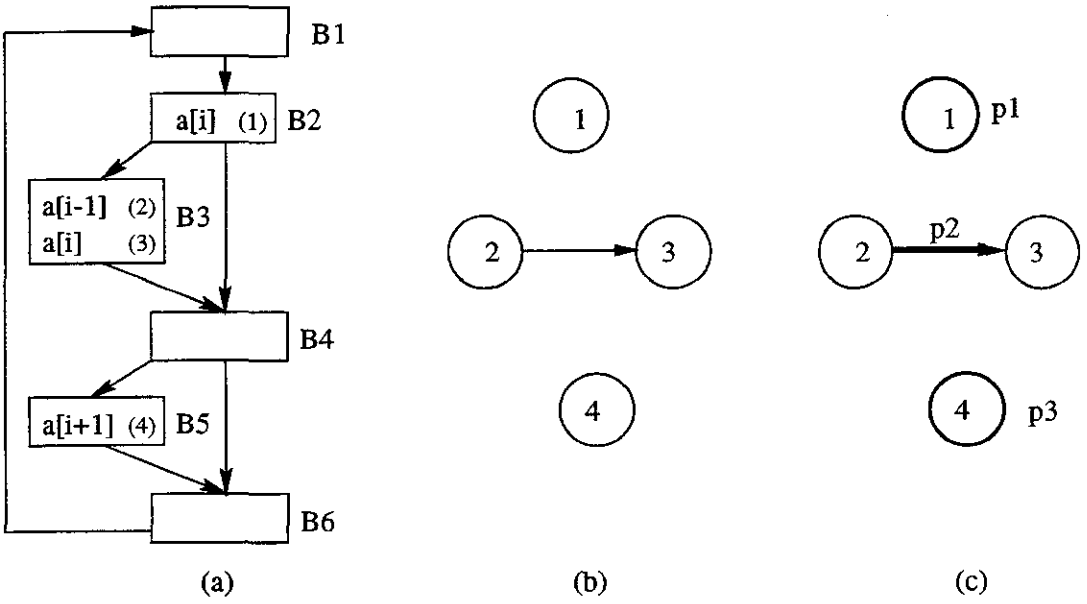


Figura 3.11: (a) Programa em CFG. (b) Indexing Graph. (c) Cobertura do IG.

Exemplo 13 Considere novamente o exemplo da Figura 3.1. Este exemplo é utilizado agora para mostrar a aplicação da SRF juntamente com a técnica de cobertura por caminhos. Para facilitar a visualização, o CFG do programa da Figura 3.1 é novamente mostrado na Figura 3.11(a). O primeiro passo no algoritmo de alocação baseado em

SRF consiste em identificar todas as referências a arrays que ocorrem no interior do laço. O algoritmo da Figura 2.2 é utilizado para isto. Em seguida, obtém-se uma lista composta pelas referências `a[i]`, `a[i-1]`, `a[i]` e `a[i+1]`, que formam os vértices do *IG*. Utilizando-se o algoritmo da Figura 3.10, as arestas do *IG* são determinadas resultando no *IG* da Figura 3.11(b). Para cada referência da Figura 3.11(a) é atribuído um número que é utilizado para rotular os vértices do *IG*. Após aplicar o algoritmo de cobertura por caminhos, obtém-se os caminhos $p1, p2$ e $p3$, ilustrados na Figura 3.11(c). Os caminhos $p1$ e $p3$ são caminhos de comprimento zero. Os caminhos resultantes da cobertura do *IG* representam as referências que podem receber o mesmo registrador de endereçamento. Desta forma, três registradores de endereçamento são requeridos. O algoritmo de análise de referências é então aplicado. O próximo passo consiste em determinar as funções-ϕ necessárias para cada caminho. Seja $ar1$ o registrador de endereçamento alocado para $p1$. O caminho $p1$ requer somente uma função-ϕ no bloco `B1`, ou seja, requer somente a ϕ-header. Como $DU_{\phi-header}$ contém somente a referência `a[i]`, que é utilizada para inicializar $ar1$ no *pre-header* do laço. A referência `a[i]` é então modificada para uma referência ao registrador $ar1$, e o modo de auto-incremento é usado para ajustar ar para a próxima iteração do laço, desde que a variável de indução é incrementada no final do laço. Seja $ar2$ o registrador de endereçamento alocado para o caminho $p2$. Duas funções-ϕ são necessárias neste caminho, uma no bloco `B1` e outra no bloco `B4`. A função-ϕ do bloco `B1`, a ϕ-header, resulta na referência `a[i-1]` desde que $DU_{\phi-header} = \{$ `a[i-1]` $\}$. Com isso, $ar2$ é inicializado com a referência `a[i-1]` no *pre-header* do laço. A função-ϕ do bloco `B4` resulta na referência `a[i-1]` desde que $UD_{\phi B4} = \{$`a[i]`, `a[i - 1]`$\}$ e $DU_{\phi B4} = \{$`a[i - 1]`$\}$, e a referência `a[i-1]` é a referência intermediária que resulta no menor custo. Determina-se então o conjunto de instruções de redirecionamento para o caminho $p2$, resultando numa única instrução adicional `ar2 += 1` no bloco `B4`. Na representação intermediária, a referência `a[i-1]` é alterada para $ar2$ fazendo uso do modo de auto-incremento, e `a[i]` é também alterada para $ar2$ fazendo uso do modo auto-decremento já que $ar2$ deverá apontar para a referência `a[i-1]` no bloco `B4`. Por fim, o registrador $ar3$ é alocado para o caminho $p3$. Duas funções-ϕ são necessárias, uma no ϕ-header e outra no bloco `B6`. A ϕ-header resulta na referência `a[i]` assim como a função-ϕ no bloco `B6`. Uma instrução de redirecionamento `ar3 += 1` é necessária no bloco `B6` para ajustar $ar3$ para a próxima iteração do laço. A Figura 3.12 mostra o código após a alocação.

Note que, se tivéssemos por exemplo uma referência `a[i+1]` no bloco `B6` da Figura 3.11(a), o mesmo ar alocado para `a[i]` em `B2` seria alocado para realizar esta referência em `B6`, utilizando para isto somente o modo de auto-incremento após realizar a referência `a[i]` em `B2`. Neste caso, o *IG* teria uma aresta entre estas referências, embora estivéssem em blocos diferentes.

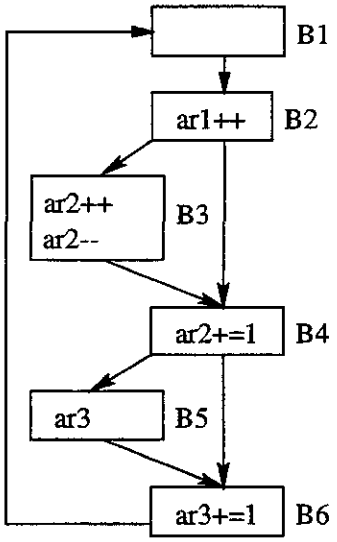


Figura 3.12: Resultado apos a alocação de registradores de endereçamento.

3.5 Algoritmo para Redução de Spilling

O algoritmo 3.4.2 realiza alocação de registradores de endereçamento no interior de um laço sem verificar se o número de registradores requeridos é maior do que o disponível na arquitetura alvo. Este é um ponto importante a ser considerado desde que a maioria dos processadores dedicados apresenta um número reduzido de registradores com modo de auto-incremento(decremento). Portanto, se o resultado apresentado pelo algoritmo 3.4.2 requerer um número de registradores maior do que os existentes na arquitetura alvo, registradores devem ser *spilled*(transbordados) para a memória. Isto resulta na inserção de instruções adicionais de `LOAD` e `STORE` para usar um registrador que foi transbordado para a memória. Como consequência, os ganhos devidos ao uso do modo de auto-incremento(decremento) são reduzidos.

Este trabalho propõe uma técnica para redução do número de registradores necessários para realizar todas as referências de um laço, sempre que a cobertura por caminhos possuir cardinalidade maior do que o número de registradores presentes na arquitetura alvo. Isto é feito através da sucessiva junção dos caminhos resultantes da cobertura, sempre que o mesmo registrador puder ser utilizado para a realização das referências em diferentes caminhos. Dois caminhos do IG são unidos se o custo da inserção de instruções de redirecionamento resultante da junção for menor do que o custo resultante da junção de outro par de caminhos. Para cada par de caminhos resultantes da operação de junção

será alocado o mesmo ar.

Definição 3.5.1 *Sejam $c_i = \{i_1, ..., i_n\}$ e $c_j = \{j_1, ..., j_n\}$ dois caminhos compostos por vértices que representam referências de um array. Seja V_i o conjunto de vértices de c_i e V_j o conjunto de vértices de c_j. Seja $V_i \cup V_j$ o conjunto de vértices dos caminhos c_i e c_j. A operação de junção de dois caminhos resulta em um conjunto de seqüências de vértices, denotado por $J(c_i \bowtie c_j) = \{\{v_1, ..., v_r\}, ..., \{v_s, ..., v_t\}\}$ tal que cada seqüência de vértices em $J(c_i \bowtie c_j)$ é formada por vértices do conjunto $V_i \cup V_j$ que ocorrem no mesmo bloco básico no CFG. Cada seqüência de vértices esta ordenada de acordo com a ordem de escalonamento das referências do array no programa.*

A operação de junção $J(c_i \bowtie c_j)$ também pode ser aplicada quando c_i ou c_j representam um conjunto de seqüências de vértices.

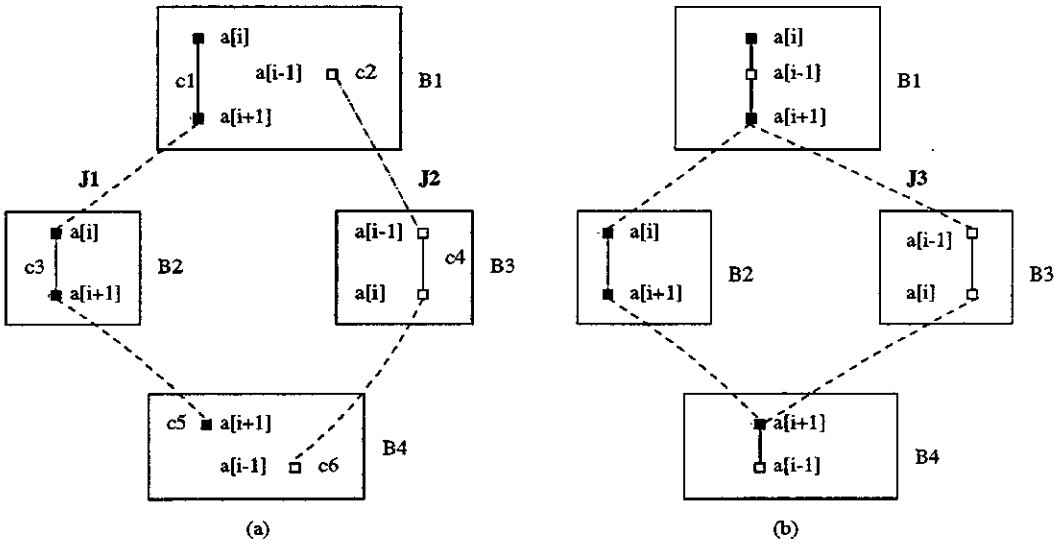


Figura 3.13: Resultado após a operação de junção dos caminhos $p1$ e $p2$.

Exemplo 14 Considere o exemplo da Figura 3.13(a). O conjunto J_1 é o resultado da operação de junção dos caminhos $c1$, $c3$ e $c5$. Já o conjunto J_2 resulta da junção dos caminhos $c2$, $c4$ e $c6$ A Figura 3.13(b) mostra o resultado da operação de junção $J(J_1 \bowtie J_2)$, que produz o conjunto de seqüências de vértices J_3. As arestas pontilhadas mostram que as seqüências de vértices pertencem ao mesmo conjunto.

O conjunto J de seqüências resultante da operação de junção de caminhos deve ser convertido para a forma SRF, já que o mesmo registrador pode ser alocado para referências

em J que estão em diferentes blocos básicos. Desta forma, instruções de redirecionamento do *ar* podem ser requeridas. Seja $S = \{v_1, ..., v_n\}$ uma seqüência de vértices pertencente a J. A seqüência S pode requerer a inserção de instruções adicionais de redirecionamento entre dois vértices v_i e $v_{i+1} \in S$. Esta inserção ocorre quando não existe uma aresta (v_i, v_{i+1}) no IG, o que causa um custo adicional no número de instruções. Este custo, somado ao custo das instruções de redirecionamento necessárias para transformar J na forma SRF, determina o custo total da operação de junção de dois caminhos.

Para transformar o resultado da operação de junção J na forma SRF, uma função-ϕ adicional, denominada *função-ϕ trivial*, é iserida após toda referência que atinge mais de uma referência em J. Por exemplo, uma função-ϕ adicional é requerida logo após a referência `a[i+1]` no final do bloco `B1` da Figura 3.13(b). Esta função-ϕ é utilizada para decidir qual a referência o registrador *ar* deve apontar no final do bloco `B1`. Para se determinar esta referência aplica-se, como para as outras funções-ϕ, o algoritmo 3.4.1, , resultando no acesso `a[i]`. Neste caso, o registrador *ar* é decrementado após a referência `a[i+1]` no final de `B1`. Além disso, a instrução `ar -=1` é inserida antes da primeira referência `a[i-1]` no bloco `B3`.

Seja $C = \{c_1, c_2, ..., c_n\}$ o cobertura de um IG G. Por motivo de simplicidade, o conjunto C conterá tanto caminhos como conjuntos de seqüências de vértices. O termo *elemento* será usado para designar um elemento pertencente a C, que pode ser um caminho ou um conjunto de seqüências. O algoritmo de junção de caminhos em C consiste nos seguintes passos:

Algoritmo 3.5.1 Algoritmo para a junção de caminhos.

1. Para cada par de elementos (c_i, c_j), tal que $1 \leq i < n$, $i+1 \leq j \leq n$, c_i e $c_j \in C$, realizar a operação de junção dos elementos da seguinte maneira:

   (a) Criar um novo conjunto J_{ij} para armazenar o conjunto de seqüências de vértices resultante da operação de junção $J(c_i \bowtie c_j)$.

   (b) Gerar um novo conjunto de vértices $V_{ij} = V_i \cup V_j$, sendo V_i o conjunto de vértices do caminho c_i, e V_j o conjunto de vértices de c_j.

   (c) Percorrer os blocos básicos em profundidade no CFG, computando as seqüências de vértices para cada bloco. Somente os blocos que possuem referências representadas por vértices em V_{ij} resultam em uma seqüência. Cada seqüência de vértices produzida por um bloco deverá ser inserida no conjunto J_{ij}.

   (d) Converter o programa para a forma SRF, inserindo as funções-ϕ necessárias, incluindo as funções-ϕ triviais.

(e) Resolver as funções-ϕ e determinar as instruções de redirecionamento necessárias, computando desta forma o custo da operação de junção.

2. Escolher o conjunto J_{ij} que resultar no menor custo possível.

3. Remover os caminhos c_i e c_j da cobertura C, que sofreram a operação de junção, e inserir o novo conjunto J_{ij} em C.

4. Realizar todos os passos acima enquanto a cardinalidade de C for maior que o número de registradores existentes e o conjunto C tiver mudado.

O algoritmo 3.5.1 de junção de caminhos deve ser chamado logo após se determinar a cobertura mínima C do IG, realizado no terceiro passo do algoritmo 3.4.2, desde que a cardinalidade de C seja maior que o número de registradores presentes na arquitetura.

Exemplo 15 Considere novamente o exemplo da Figura 3.11(a). O Exemplo 13 apresenta uma solução para a alocação de registradores de endereçamento deste laço que requer 3 registradores. No entanto, suponha que a arquitetura alvo possua somente um registrador de endereçamento. Desta forma, o código obtido na Figura 3.12 requer *spilling* de dois registradores de endereçamento. A cobertura do IG da Figura 3.11(c) é composta pelos caminhos $p1, p2$ e $p3$. Desde que os caminhos são formados por referências do mesmo array, o Algoritmo 3.5.1 é aplicado com o intutito de reduzir o número de registradores alocados. A operação de junção é então aplicada para: $J(p1 \bowtie p2)$, $J(p1 \bowtie p3)$, $J(p2 \bowtie p3)$. Para $J(p1 \bowtie p2)$, três funções-ϕ são necessárias, uma no final do bloco `B2`(trivial) e nos blocos `B1`(ϕ-header) e `B4` . Resolvendo-se as funções-ϕ, conclui-se que tanto a função-ϕ do bloco `B2` quanto a do bloco `B4` resultam na referência `a[i+1]`. Já a ϕ-header resulta na referência `a[i]`. Desta forma, a instrução de redirecionamento `ar -= 2` é requerida antes do acesso (2) no bloco `B3`. Note que neste ponto, não é feita nenhuma alteração na representação intermediária, uma vez que ainda não se sabe qual dos conjuntos resultantes possui o menor custo. Para o conjunto $J(p1 \bowtie p3)$, três funções-ϕ são requeridas, uma no final do bloco `B2` e nos blocos `B1` e `B6`, que resultam, respectivamente, nas referências: `a[i]`, `a[i+1]` e `a[i+1]` . Com isso, nenhuma instrução de redirecionamento adicional é requerida, somente o modo de auto-incremento foi necessário para o correto direcionamento do registrador. Este é o conjunto que resulta no custo mínimo. O conjunto $J(p2 \bowtie p3)$ requer três funções-ϕ, nos bloco `B1`,`B3` e `B5`, que resultam na referência `a[i+1]`. Este conjunto requer duas instruções de redirecionamento, sendo, desta forma, o conjunto que produz o maior custo. Desta forma, o conjunto $J_{13} = J(p1 \bowtie p3)$ é selecionado, os caminhos $p1$ e $p3$ são removidos da cobertura C. Desde que a arquitetura alvo possui somente um registrador, como suposto anteriormente, uma nova iteração do algoritmo é realizada. Com isso, a operação $J(J_{13} \bowtie p2)$ é realizada.

Neste caso, $J(J_{13} \bowtie p2) = \{\{a[i]\}, \{a[i-1], a[i]\}, \{a[i+1]\}\}$. Os blocos que precisam de funções-ϕ para este conjunto são: B1(ϕ-header), B2(trivial), B4 e B6. Resolvendo-se as funções-ϕ obtém-se as referências a[i] para a ϕ-header e a[i+1] para B2, B4 e B6. Com isso, somente uma instrução de redirecionamento é necessária antes do acesso (2) no bloco B3. Desde que C possui somente os elementos J_{13} e $p2$, os mesmos são removidos de C e o resultado de $J(J_{13} \bowtie p2)$ é inserido em C, tornando C um conjunto unitário, tornando falsa a condição de iteração do algoritmo. Em seguida, a representaçao intermediária é modificada, endo utilizados os modos de auto-incremento nos acessos (1) e (2) e (3), e uma instrução de redirecionamento ar -=2 é inserida antes do acesso (2) do bloco B3. Portanto, o código resultante requer somente um registrador de endereçamento. A Figura 1.1(b) mostra uma representação em alto nível do programa após a alocação.

3.6 Método Guloso de Alocação

A análise SRF também torna possível a implementação de um algoritmo de alocação baseado no método guloso. A alocação consiste basicamente na aplicação do algoritmo 3.5.1 no conjunto de referências que ocorrem no interior de um laço, sem que seja necessário construir o IG e encontrar a cobertura para o mesmo. Neste caso, o conjunto C, que representa os caminhos após a cobertura do IG no algoritmo 3.5.1, contém o conjunto de referências do laço. Ou seja, o algoritmo toma cada referência como sendo um caminho de comprimento zero e tenta alocar o mesmo registrador para o máximo de referências possível, tal como mostra o algoritmo 3.5.1. O resultado da aplicação do algoritmo guloso para o exemplo da Figura 3.11 é o mesmo que o apresentado no exemplo 15, ou seja, apenas um registrador de endereçamento é necessário para realizar todas as referências no laço. A única diferença na solução gulosa é que o conjunto $C = \{\{a[i]\}, \{a[i-1]\}, \{a[i]\}, \{a[i+1]\}\}$. Como mostrado nos resultados experimentais do Capítulo 5, o método guloso nem sempre resulta no melhor resultado, comparado ao obtido através da cobertura do IG, embora seja um algoritmo mais simples e eficiente.

3.7 Um Exemplo Real

Para ilustrar a aplicação das técnicas de alocação de registradores em um programa real, o benchmark *convenc*, pertencente ao conjunto de benchmarks da Conexant Systems Inc., é mostrado na Figura 3.14. Por se tratar de parte de uma aplicação real pertencente a esta empresa, ele é utilizado agora para mostrar o resultado após a aplicação do algoritmo 3.4.2.

Considere então o código do benchmark *convenc* na Figura 3.14. O laço na linha (7) é o primeiro a ser tratado pelo algoritmo de alocação. O registrador `ar1` é alocado para a referência `bittbl[j]`, e o modo de auto-incremento é utilizado em `ar1` para redirecioná-lo para a próxima iteração do laço. O próximo laço a ser tratado pelo algoritmo de alocação é o laço que inicia na linha (30), que é o laço mais interior. A solução para este laço, requer 4 registradores para as referências à `bittbl` nas linhas (31)-(34), já que as referências encontram-se no interior de comandos condicionais. Desde que o processador DSP da Conexant Systems Inc. possui quatro registradores de endereçamento, nenhum registrador precisa ser transbordado. Contudo, quando o laço mais externo for tratado, esse transbordamento ocorrerá. Foi verificado, tanto neste como em outros benchmarks, que em alguns casos a cobertura C do IG possui caminhos compostos pelas mesmas referências de um array. Para tirar vantagem deste fato, uma heurística para junção de caminhos é aplicada após se determinar a cobertura C. Este algoritmo basicamente varre a cobertura realizando a junção dos caminhos com referências iguais. Note que esta heurística somente é aplicada quando o número de registradores requeridos pela cobertura C é menor ou igual ao número de registradores presentes na arquitetura. Caso seja maior o algoritmo 3.5.1 é aplicado. No caso das referênicas nas linhas (31)-(34), as duas referências à `bittbl[2*i]` nas linhas (31) e (33), , que ocorrem em diferentes blocos básicos, recebem o mesmo registrador `ar1`, como mostra a Figura 3.15. De maneira análoga, as referências à `bittbl[2*i+1]` nas linhas (32) e (35) foram alocadas ao registrador `ar2`. Note que tanto o registrador `ar1` como `ar2` devem ser redirecionados para as referências na próxima iteração do laço. Este redirecionamento deve ser feito no final do laço em um bloco executado em toda iteração. Este redirecionamento é feito nas linhas (41) e (42) na Figura 3.15. Note agora que para o laço que vai das linhas (9) a (36), somente dois registradores estão disponíveis para a alocação. No entanto, três registradores são requeridos pela cobertura do IG deste laço. Um primeiro registrador para a referência `in[j]` na linha (12), outro para as referências `out[2*j]` e `out[2*j+1]` nas linhas (28) e (29), e por fim, o terceiro registrador para as referências à `out[2*j]` e `out[2*j+1]` que ocorrem no laço mais interno, nas linhas (31)-(34). Desta forma, o algoritmo 3.5.1 é aplicado, o que resulta na alocação do mesmo registrador para todas as referências do vetor `out`. A Figura 3.15 mostra o resultado após a aplicação do algoritmo de alocação. O resultado é mostrado em linguagem de alto nível para facilitar a visualização. A Figura destaca as partes do programa que sofreram alterações.

```
(1)  void Convenc(int *in, int  *out, int  n, int  *bittbl)
(2)  {
(3)     int  i, j;
(4)     unsigned int  G0, G1, D0, D1, D2, D3, D4, D5;
(5)     unsigned long  ITEMP=0, ITEMP2;
(6)
(7)     for(j=0;j<16;j++) bittbl[j]=1<<j;
(8)
(9)     for(j=0; j<n; j++)  {  /* Process N 16-bit words of input */
(10)        /* Get curr 16 bits + prev 16 bits */
(11)        ITEMP >>=16;
(12)        ITEMP2 = ((U32)in[j]) & 0x0000ffff;
(13)        ITEMP = (ITEMP2<<5) | ITEMP;
(14)        /* Undelayed input */
(15)        D5 = (U16)(0x0000ffff & ITEMP);
(16)        /* Delayed by 1 */
(17)        D4 = (U16)(0x0000ffff & (ITEMP >> 1));
(18)        /* Delayed by 2 */
(19)        D3 = (U16)(0x0000ffff & (ITEMP >> 2));
(20)        /* Delayed by 3 */
(21)        D2 = (U16)(0x0000ffff & (ITEMP >> 3));
(22)        /* Delayed by 4 */
(23)        D1 = (U16)(0x0000ffff & (ITEMP >> 4));
(24)        /* Delayed by 5 */
(25)        D0 = (U16)(0x0000ffff & (ITEMP >> 5));
(26)        G0 = D0 ^ D1 ^ D3 ^ D5;      /* G0 = D0^D1^D3^D5 */
(27)        G1 = G0 ^ D1 ^ D2 ^ D4;      /* G1 = D0^D2^D3^D4^D5 */
(28)        out[2*j]   = 0;
(29)        out[2*j+1] = 0;
(30)        for (i=0; i<8; i++)  {     /* Interleave G0 and G1 */
(31)            out[2*j] |= ((G0>>i) & 0x0001)?bittbl[2*i]:0;
(32)            out[2*j] |= ((G1>>i) & 0x0001)?bittbl[2*i+1]:0;
(33)            out[2*j+1] |= ((G0>>(i+8)) & 0x0001)?bittbl[2*i]:0;
(34)            out[2*j+1] |= ((G1>>(i+8)) & 0x0001)?bittbl[2*i+1]:0;
(35)        }
(36)    }
(37) }
```

Figura 3.14: Código do programa convenc.c.

```
(1)  void Convenc(int *in, int  *out, int  n, int  *bittbl)
(2)  {
(3)     int  i, j;
(4)     unsigned int  G0, G1, D0, D1, D2, D3, D4, D5;
(5)     unsigned long  ITEMP=0, ITEMP2;
(6)     register *ar1,*ar2,*ar3,*ar4;
(7)     ar1 = &bittbl[0];
(8)     for(j=0;j<16;j++) *ar1++ = 1<<j;
(9)
(10)    ar3 = &in[0];
(11)    ar4 = &out[0];
(12)    for(j=0; j<n; j++)  {  /* Process N 16-bit words of input */
(13)        /* Get curr 16 bits + prev 16 bits */
(14)        ITEMP >>=16;
(15)        ITEMP2 = ((U32)*ar2++) & 0x0000ffff;
(16)        ITEMP = (ITEMP2<<5) | ITEMP;
(17)        /* Undelayed input */
(18)        D5 = (U16)(0x0000ffff & ITEMP);
(19)        /* Delayed by 1 */
(20)        D4 = (U16)(0x0000ffff & (ITEMP >> 1));
(21)        /* Delayed by 2 */
(22)        D3 = (U16)(0x0000ffff & (ITEMP >> 2));
(23)        /* Delayed by 3 */
(24)        D2 = (U16)(0x0000ffff & (ITEMP >> 3));
(25)        /* Delayed by 4 */
(26)        D1 = (U16)(0x0000ffff & (ITEMP >> 4));
(27)        /* Delayed by 5 */
(28)        D0 = (U16)(0x0000ffff & (ITEMP >> 5));
(29)        G0 = D0 ^ D1 ^ D3 ^ D5;      /* G0 = D0^D1^D3^D5 */
(30)        G1 = G0 ^ D1 ^ D2 ^ D4;      /* G1 = D0^D2^D3^D4^D5 */
(31)        *ar4++   = 0;
(32)        *ar4     = 0;
(33)        ar1      = &bittbl[0];
(34)        ar2      = &bittbl[1];
(35)        for (i=0; i<8; i++)  {     /* Interleave G0 and G1 */
(36)            ar4 -= 1;
(37)            *ar4 |= ((G0>>i) & 0x0001)?*ar1:0;
(38)            *ar4++ |= ((G1>>i) & 0x0001)?*ar2:0;
(39)            *ar4 |= ((G0>>(i+8)) & 0x0001)?*ar1:0;
(40)            *ar4 |= ((G1>>(i+8)) & 0x0001)?*ar2:0;
(41)            ar1 += 1;
(42)            ar2 += 1;
(43)        }
(44)    }
```

Figura 3.15: Resultado em alto nível após a alocação de registradores. Modificações em destaque.

Capítulo 4

Análise de Desempenho

A grandeza não consiste em receber honras, mas em merecê-las.

Aristóteles

Neste capítulo são apresentados os resultados experimentais obtidos após a aplicação das técnicas apresentadas nos Capítulos 2 e 3.. As técnicas apresentadas foram implementadas em um compilador comercial. A Seção 4.1 descreve os benchmarks utilizados para medir o desempenho. A Seção 4.2 descreve sucintamente o compilador adotado. A seção 4.3 apresenta as melhorias de desempenho obtidas quando comparadas à técnica de alocação de registradores convencional baseada em coloração do grafo de interferência [29, 1].

4.1 Benchmarks Adotados

Desde que processadores embutidos são em grande parte utilizados para a solução de problemas específicos, não existe um conjunto de benchmarks públicos expressivo. Com o objetivo de se medir o resultado após a aplicação das técnicas de alocação de registradores de endereçamento apresentadas, um conjunto de benchmarks foi adotado. Os benchmarks adotados incluem kernels de programas reais pertencentes à Conexant Systems Inc., outros pertences à Texas Instruments, e programas do benchmark DSP Stone [35]. Estes benchmarks representam o núcleo de muitas aplicações que executam em processadores DSPs, o que os tornam atrativos para efeito de avaliação das técnicas aqui apresentadas.

4.2 Compilador Utilizado

O compilador utilizado para a implementação das técnicas deste trabalho é um compilador de produção pertencente à Conexant Systems. A arquitetura alvo deste compilador é o processador Countach40 pertencente à Conexant Systems Inc que é um DSP utilizado no projeto de modems, cartões grafos, cartões de áudio, GPS e outros sistemas portáteis. Este compilador é um compilador para a linguagem C que realiza todas as otimizações relevantes descritas em [1], como Eliminação de expressões comuns, Eliminação de Variáveis de Indução, Remoção de Código Invariante, etc. Este processador possui 4 registradores de endereçamento com modo de auto-incremento(decremento). Um simulador para o processador Countach40, pertencente à mesma empresa, foi utilizado para medir o número de ciclos, e desta forma calcular o speedup resultante da aplicação das soluções apresentadas.

4.3 Análise dos Resultados

A Tabela 4.1 mostra o número de ciclos resultantes após a aplicação das técnicas de alocação apresentadas nos benchmarks adotados. Os resultados experimentais mostram uma média de desempenho de 11.3% quando o algoritmo SRF baseado no IG é comparado com o algoritmo de alocação original, que não realiza a alocação de um mesmo registrador de endereçamento entre referências de array, usando o algoritmo priority-based register coloring [29]. O algoritmo de alocação original realiza a alocação baseado em coloração, utilizando o suporte de todas as otimizações tradicionais para otimizar o cálculo do endereço de um elemento de array, como eliminação de sub-expressões comuns, eliminação de variáveis de indução e remoção de código invariante de laço. Quando a variação gulosa do algoritmo SRF é utilizada, um speed-up médio de 11% é obtido. Note que em um grande número de casos a solução gulosa resulta no mesmo speed-up que o algoritmo SRF baseado na cobertura do IG. No entanto, existem casos em que a solução gulosa é menos eficiente que a SRF exata. Na Tabela 4.1, a primeira coluna lista todos os benchmarks utilizados. A segunda coluna consiste no número de ciclos obtidos após a aplicação do algoritmo original, Priority-Based coloring, para o conjunto de benchmarks. A terceira coluna mostra o número de ciclos resultantes após a aplicação da técnica de SRF baseada no IG. O speedup de cada programa também é mostrado. Os resultados mais expressivos, como nos benchmarks *n_complex_updates* e *n_real_updates*, ocorrem devido à grande ocorrência de referências a elementos de arrays em laços aninhados nestas aplicações. A útima coluna mostra os resultados após a aplicação do algoritmo guloso baseado em SRF, SRF Algorithm Greedy. O algoritmo guloso produz resultados tão bons quanto aos resultados da solução usando IG.

Observe que foram utilizados na validação e avaliação de nossas técnicas um compilador de produção e partes de programas reais.

Em geral, melhorias no tempo de execução acima de 5% são consideradas boas, o que mostra que o ganho no desempenho de execução obtido após a aplicação das técnicas deste trabalho, aproximadamente 11%, é bastante satisfatório.

	Priority-Based coloring	SRF Algorithm IG Based		SRF Algorithm Greedy	
Program	cycles	cycles	Speedup	cycles	Speedup
convenc	4331	3943	9%	3967	9%
convolution	1220	1042	17%	1042	17%
dot_product	165	160	3%	161	2%
biquad_N_sections	1380	1218	13%	1218	13%
fir_array	1471	1263	16%	1263	16%
fir2dim	7684	6728	14%	6728	14%
lms_array	2276	1919	18%	1919	18%
mat1x3	1202	1113	7%	1130	6%
matrix1	34657	30520	13%	30520	13%
n_complex_updates	2985	2336	27%	2336	27%
n_real_updates	1855	1452	27%	1452	27%
fft	173931	165549	5%	167549	3%
autcor	179633	167238	7%	167328	7%
fir8	280324	256476	9%	256476	9%
latsynth	3115	3050	2%	3059	1%
fir_lms2	3454	3317	4%	3317	4%
latanal	703662	685162	2%	691662	1%

Tabela 4.1: Resultados após a aplicação de AIA

Capítulo 5

Conclusões e Trabalhos Futuros

O bom não é ser importante, o importante é ser bom.

Padre Roque Schneider

Neste capítulo são apresentados um resumo da dissertação, as principais contribuições (Seção 5.1) e também algumas sugestões para trabalhos futuros (Seção 5.2).

5.1 Resumo e Contribuições

Este trabalho apresentou três contribuições originais a solução do problema de alocação de registradores de endereçamento a referências de arrays em laços. Primeiramente, foi proposta uma extensão, para o caso multidimensional, da técnica de alocação sugerida por Araujo et al [3], onde a alocação é modelada através da solução de um problema de cobertura em grafos. Como segunda contribuição, este trabalho propõe uma extensão da estratégia em [3] para o caso quando as referências ocorrem em blocos básicos diferentes no interior de um laço. Isto foi feito usando uma variação da forma SSA, denominada de *Single Reference Form* (SRF), para representar referências a arrays no programa. Finalmente, foi estudado um algoritmo para a redução do número de *spilling* de registradores de endereçamento, nos casos em que o número de registradores alocados pela solução em [3] for maior que o número de registradores existentes na arquitetura alvo. Para isto, definiu-se um operador de junção de caminhos e uma heurística baseada neste.

5.2 Trabalhos Futuros

A representação de um programa em SRF, proposta nesta tese, abre espaço para uma nova maneira de se realizar alocação de registradores de endereçamento a apontadores. Uma aplicação imediata desta abordagem seria a tradução de referências a arrays no programa-fonte para acessos via apontadores, usando aritmética de ponteiros. Note que a única séria restrição ao programa fonte imposta pela técnica descrita nesta trabalho , diz respeito a ordem relativa com que as referências aparecem no programa. Certamente, esta somente é conhecida ao final do escalonamento das instruções feito no *back-end* do compilador. No entanto, pode ser desejável forçar o escalonamento das referências reescrevendo ainda programa fonte, de modo a garantir que a ordem entre as referências esteja definida antes do início da compilação. Uma vez isto sendo feito, seria possível então converter referências para acessos via apontadores, induzindo o compilador a alocar um número menor de registradores de endereçamento e a minimizar o número de instruções de redirecionamento necessárias. Note que, a princípio, tudo pode ser feito no programa-fonte, independentemente do compilador e da aquitetura alvo. Uma avaliação precisa desta estratégia está sendo estudada, e os seus resultados serão apresentados futuramente.

Referências Bibliográficas

[1] V. V. Aho, R. Sethi, e J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986. Reading, MA.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge, 2 edição, 1998. Cambridge, United Kingdom.

[3] G. Araujo, A. Sudarsanam, e Malik S. Instruction set design and optimizations for address computation in DSP processors. Em *9th International Symposium on Systems Synthesis*, pp. 31–37. IEEE, November de 1996.

[4] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software Practice and Experience*, 22(2):101, February de 1992.

[5] F. Boesch e J Gimpel. Covering the points of a digraph with point-disjoint paths and its aplications to code optimization. *Journal of the ACM 24, 2 (April)*, pp. 192–198, 1977.

[6] D.G. Bradlee, Eggers S.J., e R.R. Henry. Integrating register allocation and instruction scheduling for RISCs. Em *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, April de 1991.

[7] P. Briggs, K. Cooper, K. Kennedy, e L. Torczon. Coloring heuristics for register allocation. Em *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation*, pp. 98–105, June de 1982.

[8] D. Callahan e B Koblenz. Register allocation via hierarchical graph coloring. Em *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 192–202, June de 1991.

[9] G. Chaitin. Register allocation and spilling via graph coloring. Em *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pp. 98–105, June de 1982.

[10] F.C. Chow e J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, October de 1990.

[11] T.H. Cormem, C.E. Leiserson, e R.L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill, Boston, 1990.

[12] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, e F.K. Zadeck. An efficient method of computing static single assignment form. Em *Proc. of the ACM POPL'89*, pp. 23–25, 1989.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, e F. Kenneth Zadeck. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No 4:451–490, 4 (October) de 1991.

[14] Editorial. The Future of Computing. *The Economist*, pp. 79–81, Set. de 1998.

[15] M. Garey e D. Johnson. Computers and intractability. *W. H. Freeman and Company*, 1979. New York.

[16] Catherine Gebotys. DSP address optimization using a minimum cost circulation technique. Em *Proceedings of the International Conference on Computer-Aided Design*, pp. 100–103. IEEE, November de 1997.

[17] J.R. Goodman e A.W. Hsu. Code scheduling and register allocation in large basic blocks. Em *Proceedings of the 1988 Conference on Supercomputing*, pp. 442–452, July de 1988.

[18] R Gupta, M.L. Soffa, e D. Ombres. Efficient register allocation via coloring using clique separators. *ACM Trans. Programming Language and Systems*, 16(3):370–386, May de 1994.

[19] John L. Hennessy e David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufman, 2nd edição, September de 1997.

[20] C. Y. Hitchcock III. Addressing modes for fast and optimal code generation. *Ph.D thesis, Carnegie-Mellon University*, 1993. Pittsburg, PA.

[21] J. Hopcroft e R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput, 2,4 (December)*, pp. 225–230, 1973.

[22] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, e G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. Em P. Marwedel e G. Goossens, editores, *Code Generation for Embedded Processors*, capítulo 5, pp. 85–102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[23] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, June de 1997.

[24] R. Leupers e F. David. A uniform optimization technique for offset assignment problems. *Int. Symp. on System Synthesis (ISSS)*, 1998. Hsinchu/Taiwan.

[25] R. Leupers e P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. Em *Proceedings of the International Conference on Computer-Aided Design*, 1996.

[26] Rainer Leupers, Anupam Basu, e Peter Marwedel. Optimized array index computation in DSP programs. Em *Proceedings of the ASP-DAC.* IEEE, February de 1998.

[27] S. Liao, S. Devadas, K. Keutzer, e A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18:235–253, May de 1996.

[28] P. Marwedel e G. Goossens. Code generation for embedded processors. *Kluwer Acad Pub*, 1995.

[29] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[30] P. G. Paulin, C. Liem, T. C. May, e S. Sutarwala. CodeSyn: A Retargetable Code Synthesis System. Em *Proceedings of the 7th International High-Level Synthesis Workshop*, Spring de 1994.

[31] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* Tese de Doutorado, Princeton University, May de 1998.

[32] A. Sudarsanam, S. Liao, e S. Devadas. Analysis and evaluation of address arimethic capabilities in custom dsp architectures. Em *Proc. of the ACM/IEEE Design Automation Conference*, 1997.

[33] N. Sugino, H. Miyazaki, S. Iimure, e A. Nishihara. Improved code optimization method utilizing memory addressing operation and its application to dsp compiler. Em *Proc. of the Int. Symposium on Circuits and Systems (ISCAS)*, 1997.

[34] B. Wess e M. Gotschlich. Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem. Em *Proceedings of the International Symposium on Circuits and Systems*, 1997.

[35] V. Zivojnovic, J.M. Velarde, e C. Sclåager. DSPstone, a DSP benchmarking methodology. Technical report, Aachen University of Technology, August de 1994.