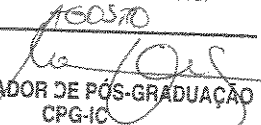


Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Desirée Leopoldo da Silva Ottoni
e aprovada pela Banca Examinadora.
Campinas, 25 de AGOSTO de 2004

COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

**Algoritmos Para Alocação de Pilha
Baseados em União de Variáveis Para DSPs**

Desirée Leopoldo da Silva Ottoni

Dissertação de Mestrado

Algoritmos Para Alocação de Pilha Baseados em União de Variáveis Para DSPs

Desirée Leopoldo da Silva Ottoni¹

Março de 2004

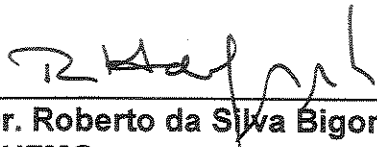
Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Tomasz Kowaltowski
Instituto de Computação – UNICAMP
- Prof. Dr. Roberto da Silva Bigonha
Departamento de Ciência da Computação – UFMG
- Prof. Dr. Rodolfo Jardim Azevedo (suplente)
Instituto de Computação – UNICAMP

¹Auxílio financeiro da FAPESP (processo 01/12762-5).

TERMO DE APROVAÇÃO

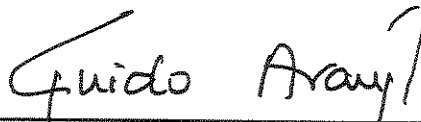
Tese defendida e aprovada em 19 de março de 2004, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Roberto da Silva Bigonha
DCC – UFMG



Prof. Dr. Tomasz Kowaltowski
IC - UNICAMP



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

UNIDADE	BL
Nº CHAMADA	T/VV/Camp
	Ot8a
V	EX
TOMBO, BC/	61019
PROC.	16.117-07
C	<input type="checkbox"/>
D	<input checked="" type="checkbox"/>
PREÇO	21,00
DATA	19.11.07
Nº CPD	

B.0 Id 332674

**FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP**

Otoni, Desirée Leopoldo da Silva

Ot8a Algoritmos para alocação de pilha de execução baseados em união de variáveis para DSPs / Desirée Leopoldo da Silva Otoni -- Campinas, [S.P. :s.n.], 2004.

Orientador : Guido Costa Souza de Araújo

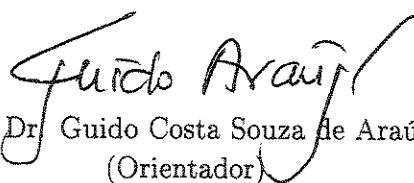
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

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

Algoritmos Para Alocação de Pilha Baseados em União de Variáveis Para DSPs

Este exemplar corresponde à redação final da
Dissertação devidamente corrigida e defendida
por Desirée Leopoldo da Silva Ottoni e apro-
vada pela Banca Examinadora.

Campinas, 19 de março de 2004.


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

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

8260328

© Desirée Leopoldo da Silva Ottoni, 2004.
Todos os direitos reservados.

Resumo

Nos últimos anos, uma classe importante de aplicações em telecomunicações e multimídia tem despertado um grande interesse no projeto e pesquisa de processadores dedicados, em particular de DSPs². Além de desempenho, estas aplicações demandam baixo consumo de potência e custo reduzido. Com o propósito de atender a esta demanda, projetistas de DSPs precisam especializar suas arquiteturas com unidades funcionais dedicadas. Devido a rigorosas restrições de projeto, é comum encontrar DSPs com poucos registradores de propósito geral e modos de endereçamento restritos, baseados em unidades especializadas no cálculo de endereços de memória. Por serem arquiteturas irregulares, as otimizações de código existentes nos compiladores para processadores de propósito geral não são eficientes para DSPs. Isto resultou em um aumento no interesse por pesquisa de técnicas de otimizações para estes processadores.

Esta dissertação propõe duas novas técnicas de otimização de código para o problema de *Offset Assignment*(OA). Uma solução para OA visa encontrar uma disposição das variáveis automáticas de um programa na memória, de forma a minimizar o uso de instruções explícitas de endereçamento, obtendo assim um código de melhor desempenho. Este tipo de otimização é um dos problemas centrais de compilação para DSPs, dado que grande parte das instruções geradas para estes processadores é de endereçamento.

Uma extensa revisão bibliográfica sobre *Offset Assignment* é apresentada nesta dissertação. Além disso, são propostos dois novos algoritmos que resolvem variações deste problema: a heurística CSOA, que resolve o problema de *Simple Offset Assignment*, e a heurística CGOA, que resolve o problema de *General Offset Assignment*. As duas heurísticas utilizam informações de longevidade das variáveis de modo a realizar união seletiva de variáveis na memória, resultando em uma melhor utilização de modos de endereçamento de auto-incremento/decremento. Além das duas técnicas propostas, foram implementadas outras quatro técnicas existentes na literatura. Uma análise comparativa, baseada num conjunto de experimentos usando o benchmark Mediabench, revelou a superioridade de CSOA e CGOA sobre os outros métodos.

²Do inglês: *Digital Signal Processors*

Abstract

In recent years, an important class of applications in telecommunication and multimedia has created a large interest in the design and research of dedicated processors, specially Digital Signal Processors (DSPs). In addition to performance, these applications demand low power consumption and reduced cost. In order to achieve these goals, DSP designers need to specialize the architecture with dedicated functional units. Due to their stringent design constraints, it is common to find DSPs containing very few general-purpose registers, and restricted addressing modes, typically based on specialized address generation units. Given their irregular architectures, compiler code optimization techniques for general-purpose processors are not efficient for DSPs. This has resulted in an increasing interest in the research of optimization techniques target to such processors.

This dissertation proposes two novel code optimization techniques for the Offset Assignment (OA) problem. A solution to OA aims at finding a memory layout for automatic variables in a program, such that the use of explicit memory addressing instructions is minimized, thus increasing the performance of the resulting code. This type of optimization is one of the central problems in compilation for DSPs, as address computation accounts for a large share of the instructions generated for these processors.

A long survey on OA is presented in this dissertation. Moreover, two new algorithms to solve variations of OA are proposed: the CSOA heuristic, to solve the Simple Offset Assignment problem; and the CGOA heuristic, which solves the General Offset Assignment. Both techniques use liveness information to perform selective coalescing of variables in memory, resulting in an improved use of auto-increment/decrement addressing modes. In addition to the two proposed algorithms, four other techniques from the literature have been implemented. A comparative analysis, based on a set of experiments using the MediaBench benchmark, has revealed the superiority of CSOA and CGOA with respect to the other methods.

Agradecimentos

Agradeço ao meu esposo, Guilherme, todo o amor, apoio, compreensão e incentivo durante os 12 anos em que estamos juntos. Agradeço também a coragem que me dá para vencer todos os obstáculos da nossa vida.

Aos meus pais, Norma e Enedino, todo o afeto, apoio e incentivo ao estudo que me ajudaram a chegar até aqui. Principalmente à minha mãe que me ensinou desde pequena a gostar dos livros e de aprender. :-)

Ao meu orientador, a amizade, os conselhos, o incentivo. Agradeço também porque, muitas vezes, acreditou em mim até mais do que eu própria. Certamente, vou sentir muitas saudades.

Agradeço também a minha madrinha e tia, Nilza, o carinho e a presença em todos os momentos da minha vida.

Aos professores Tomasz e Roberto que aceitaram revisar este trabalho.

À FAPESP e à FAEP-UNICAMP, o suporte financeiro concedido durante este Mestrado.

A todos os grandes amigos que fiz em Campinas, que deixaram a minha estada aqui muito mais divertida. Principalmente aos meus amigos Rodolfo, Juliana, Sandro, Narhi e Michele. Espero poder sempre rever todos vocês.

Ao professor Rainer Leupers, que cedeu seu compilador para que eu pudesse realizar este trabalho. Além de ter sido incansável, quando muitas vezes o perturbei com minhas dúvidas.

A todos os meus professores, do Instituto de Educação Juvenal Miller, Colégio Técnico Industrial, da Fundação Universidade Federal de Rio Grande e da UNICAMP visto que contribuíram para minha formação, e foram muito amigos.

A todos vocês, do fundo do meu coração, o meu muito-obrigado, e um grande abraço.

Desirée

Sumário

Resumo	vii
Abstract	viii
Agradecimentos	ix
1 Introdução	1
2 Unidades de Geração de Endereços em DSPs	4
3 O problema de OA	6
3.1 Conceitos Básicos	7
3.2 <i>Simple Offset Assignment</i> (SOA)	8
3.3 <i>General Offset Assignment</i> (GOA)	14
3.4 Alocação de Referências a Vetores	16
4 União de Variáveis no Problema de SOA (CSOA)	18
4.1 Análise de Complexidade do Algoritmo CSOA	22
4.2 Exemplo de CSOA	22
5 União de Variáveis no Problema de GOA (CGOA)	25
5.1 Análise de Complexidade de CGOA	27
5.2 Exemplo de Particionamento em CGOA	27
6 Resultados Experimentais	30
6.1 Resultados Obtidos para SOA	32
6.2 Resultados Obtidos para GOA	35
7 Conclusão e Trabalhos Futuros	40
Bibliografia	42

Lista de Tabelas

6.1	Custo de offset relativo ao custo obtido pela heurística SOA-OFU.	32
6.2	Custo de offset relativo ao custo obtido pela heurística SOA-OFU.	33
6.3	Tamanho da pilha relativo ao tamanho da pilha utilizada pelos métodos que não fazem união de variáveis.	34
6.4	Percentagem de variáveis temporárias.	34
6.5	Custo de offset relativo ao custo obtido pela heurística SOA-OFU, utilizando 2 ARs.	35
6.6	Custo de offset relativo ao custo obtido pela heurística SOA-OFU, utilizando 4 ARs.	36
6.7	Custo de offset relativo ao custo obtido pela heurística SOA-OFU, utilizando 8 ARs.	37
6.8	Percentagem de memória resultante do método CGOA relativo ao tamanho de memória usado pelos métodos que não utilizam união de variáveis. . . .	37
6.9	Custo de offset relativo ao custo obtido pela heurística SOA-OFU, considerando 2 ARs.	38
6.10	Custo de offset relativo ao custo obtido pela heurística SOA-OFU, considerando 4 ARs.	39
6.11	Custo de offset relativo ao custo obtido pela heurística SOA-OFU, considerando 8 ARs.	39

Lista de Figuras

2.1	Modelo de uma AGU	5
3.1	(a) Trecho de um programa em C (b) Seqüência de acessos (c) grafo de acessos (d) <i>offset assignment</i> (e) Código em <i>assembly</i> para o trecho do programa.	7
3.2	(a) seqüência de acessos (b) grafo de acessos com arestas pertencentes ao caminho retornado pelo algoritmo de Liao et al em negrito (c) grafo de acessos com arestas pertencentes ao caminho retornado pelo algoritmo de Sudarsanam et al em negrito, e as arestas induzidas em pontilhado.	10
3.3	Casos possíveis a serem tratados pela heurística que calcula o número de instruções de endereçamento economizadas quando o vértice j é unido ao vértice i	13
3.4	(a) Seqüência de acessos e grafo de acessos (b) Subseqüência de acessos e grafo gerado por b, c (c) Subseqüência de acessos e grafo gerado por a, d, e	15
4.1	(a) Grafo de Acessos (b) Grafo de acessos depois da união das variáveis u e v	21
4.2	(a) Um fragmento de código em C, com informações de vida em cada ponto do programa. (b) O grafo de interferência das variáveis. (c) A seqüência de acesso deste fragmento de programa. (d)-(j) O grafo de acessos resultante depois de cada iteração do algoritmo. (k) O <i>layout</i> da memória. As arestas selecionadas estão em destaque.	23
5.1	(a) Grafo de Interferência (b) Variáveis ordenadas em ordem decrescente dos seus números de interferência (c)-(h) Grafo de interferência das partições durante o particionamento (i) Grafos de interferência das partições após a execução do particionamento.	29

Capítulo 1

Introdução

Nas últimas décadas, houve uma grande mudança no uso e, conseqüentemente, no projeto dos processadores. Durante praticamente todos os anos 80, os processadores mais utilizados eram baseados em arquiteturas de propósito geral destinadas principalmente a PC's¹. O início da década de noventa testemunhou o surgimento de uma nova classe de aplicações comuns ao nosso dia-a-dia, tais como: telefones celulares, GPSs, *palmtops*, etc.

Por serem aplicações de tempo real muitas vezes baseadas em sistemas embutidos, geralmente à bateria, estas aplicações demandam requisitos de projeto muito mais rigorosos que os PCs, tais como: alto desempenho, baixo custo, e baixo consumo de potência [2]. Para alcançar estas metas, os projetistas precisam especializar a arquitetura do processador, de modo a economizar área de silício e potência consumida sem perder desempenho. Ou seja, unidades funcionais não especializadas deram lugar a unidades funcionais altamente especializadas, destinadas a operações comuns ao domínio de aplicação em questão. Isto tipicamente resulta em arquiteturas especializadas e irregulares, o que dificulta muito a geração de código eficiente para estes processadores.

Uma das classes de aplicações que surgiram nos últimos anos foi a voltada ao processamento de áudio e vídeo em telecomunicações e multimídia. Os processadores utilizados neste domínio passaram a se chamar *Digital Signal Processors* (DSPs). Visto que os DSPs possuem arquiteturas muito irregulares, as técnicas de geração e otimização de código para processadores de propósito geral [1] não são suficientes para gerar código de qualidade para estes processadores [13]. Os códigos gerados pelos compiladores para DSPs são entre 400% a 1000% piores (em termos de tamanho e de desempenho) do que os códigos programados diretamente em *assembly* [23]. Com isso, os DSPs são usualmente programados usando a linguagem *assembly*.

Com o crescimento do mercado de aplicações para estes processadores, os grandes períodos de tempo gastos para programar e depurar suas aplicações tornaram-se inacei-

¹Do inglês, *Personal Computer*.

táveis. Com isso, a demanda por compiladores para linguagens de alto nível que sejam capazes de gerar códigos eficientes para DSPs está cada vez maior. Por este motivo, novas técnicas de otimização para DSPs têm sido estudadas [2, 13].

O estudo de técnicas que minimizem o número de instruções de endereçamento para esta classe de processadores tornou-se relevante, dado que o tamanho do código associado às instruções de endereçamento pode representar mais de 50% de todos os bits de um programa. Foi mostrado por Udayanarayanan e Chakrabarti em [27] que, para um conjunto de programas do benchmark MediaBench [7], compilado para a família Motorola DSP56000, mais do que 55% das instruções envolvem registradores de endereço. Outras características de boa parte dos DSP's que justificam o estudo de técnicas de otimização para endereçamento são: o fato de eles não possuírem modos de endereçamento mais elaborados, tais como o modo de endereçamento indexado, sendo utilizado modo de endereçamento indireto, e a escassez de registradores de propósito geral nestes processadores, o que resulta em um grande uso da memória para armazenamento das variáveis.

O trabalho descrito nesta dissertação teve como principal objetivo o estudo e a proposta de novas soluções para o problema conhecido como *Offset Assignment* (OA). Resolver este problema consiste em associar uma posição na pilha para cada variável escalar que será alocada na pilha. Este problema ocorre em DSPs que não possuem modo de endereçamento indexado e que possuem poucos registradores de propósito geral. Devido a essas restrições, eles empregam endereçamento indireto com aritmética de auto-incremento e auto-decremento, usando registradores de endereço (ARs²) para apontar para seus endereços de memória. Para atualizar os ARs com os próximos endereços de memória que serão acessados, é preciso realizar cálculos de endereço. Se para estes cálculos não for possível utilizar instruções com auto-incremento e auto-decremento, que são instruções onde a computação do endereço é feita em paralelo com outra operação, serão introduzidas instruções explícitas de aritmética de endereço, acarretando em um aumento no tamanho do código e conseqüentemente em uma queda no desempenho. Visto que a disposição das variáveis na memória influencia muito na possibilidade de uso de instruções de auto-incremento e auto-decremento, o problema OA visa encontrar uma disposição das variáveis escalares e locais na memória de forma que se use ao máximo este tipo de instruções, obtendo um código muito mais enxuto.

Por influenciar o número de instruções geradas, a disposição das variáveis na memória acaba afetando também o custo final do sistema. Dado que grande parte destes processadores é utilizada em equipamentos onde existe muita limitação de tamanho, como, por exemplo, em telefones celulares, DSPs são projetados com severas limitações de memória. Deste modo, uma diminuição no tamanho de código gerado implica em uma memória menor e conseqüentemente em economia de área de silício.

²Do inglês *Address Register*

Esta dissertação faz um estudo das técnicas já conhecidas de *Offset Assignment*, e propõe duas novas soluções para este problema. Ela está organizada em sete capítulos, da seguinte forma:

- Capítulo 2: Este capítulo apresenta uma introdução às unidades de cálculo de endereço típicas dos DSPs.
- Capítulo 3: Este capítulo introduz o problema de *Offset Assignment* (OA), algumas definições importantes para o entendimento das heurísticas empregadas para resolvê-lo, e uma descrição das técnicas já existentes.
- Capítulo 4: Este capítulo descreve a heurística proposta nesta dissertação para a solução do problema de *Simple Offset Assignment* (SOA). Denomina-se SOA o problema de OA no qual se tem um único registrador de endereço. O trabalho descrito neste capítulo foi apresentado no *7th International Workshop on Software and Compilers for Embedded Systems - SCOPES 2003* [20] em setembro de 2003, realizado em Viena na Áustria. Este artigo foi agraciado com um *Best Paper Award* da conferência.
- Capítulo 5: Neste capítulo é apresentada a técnica proposta para a resolução do problema de *General Offset Assignment* (GOA), que é o problema de OA com múltiplos registradores de endereço.
- Capítulo 6: O capítulo 6 descreve como os experimentos foram realizados, e faz uma comparação entre os resultados obtidos pelas técnicas propostas nesta dissertação e as descritas na literatura.
- Capítulo 7: Por fim, este capítulo apresenta algumas idéias deixadas para trabalhos futuros, e as conclusões deste trabalho.

Capítulo 2

Unidades de Geração de Endereços em DSPs

Uma vez que desempenho é um fator muito importante para os DSPs, e que instruções de endereçamento representam grande parte do código de um programa, unidades funcionais especializadas na geração de endereços foram incorporadas a estes processadores. Estas unidades são chamadas de Unidades de Geração de Endereço (AGUs¹). Elas são capazes de realizar a computação de endereços rapidamente e em paralelo com outras operações de máquina, sem utilizar recursos do *datapath*. Este tipo de computação é chamada de auto-incremento quando a computação feita é uma soma e auto-decremento quando é uma subtração. Este tipo de operação aritmética é embutida como parte de outras instruções que utilizam o *datapath*, eliminando a necessidade de instruções aritméticas explícitas quando for possível utilizar auto-incremento/decremento.

Modos de endereçamento absoluto não são usados no projeto de instruções para DSP's, privilegiando-se instruções com endereçamento indireto. Isto é devido ao fato de que o endereçamento indireto reduz o número de bits usados para codificar instruções, permitindo ainda que registrador de endereço seja atualizado com o endereço do dado usado na próxima instrução, em paralelo à execução da operação corrente.

A AGU de um DSP padrão possibilita a existência de três modos de endereçamento de instrução: indireto, modular e aritmética reversa de endereço. Como para o problema de OA só o modo de endereçamento indireto é relevante, somente este modo será abordado.

No modo de endereçamento indireto, o endereço de um dado é calculado adicionando ou subtraindo um *offset* de um registrador de endereço (AR). Neste modo de endereçamento, o registrador especificado (AR) não contém o dado necessário, mas sim o endereço de memória onde o dado se encontra. Como já foi dito anteriormente, o cálculo do endereço é feito subtraindo ou somando um *offset* de um registrador de endereço (AR).

¹Do inglês: *Address Generation Units*

Tanto o número de registradores de endereço quanto o *offset* podem variar de processador para processador. Um valor de *offset* para auto-incremento e auto-decremento muito utilizado é 1, visto que há um grande número de acessos sequenciais nos programas escritos para DSPs. Mas há processadores que, como o ADSP-210x, possuem 4 registradores de endereço e *offset* de até 4.

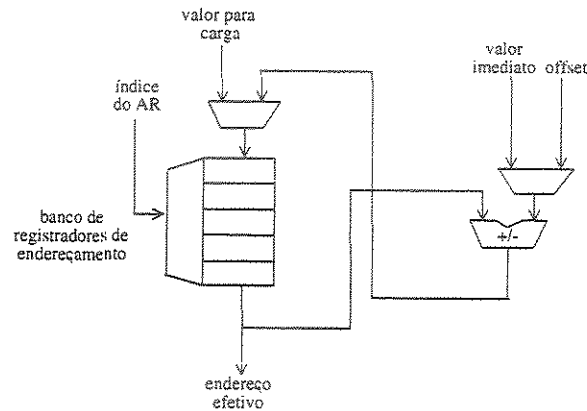


Figura 2.1: Modelo de uma AGU

Na Figura 2.1, pode-se observar uma AGU com um banco de registradores de endereço. Neste tipo de AGU é possível fazer operações de auto-incremento e decremento (em paralelo à computação), operações de subtração ou soma de um imediato a um AR e operações de carga de um AR. Por necessitarem de mais bits para serem codificadas, as operações com imediato e de carga não podem ser codificadas juntamente com outras operações em uma única instrução. Assim, durante a sua execução elas deixam ociosos recursos do *datapath*, consumindo ciclos adicionais de CPU.

É possível encontrar também, em algumas AGUs, um banco de registradores chamados de *modify registers* (MRs). Estes registradores são responsáveis por conter *offsets*. Operações do tipo soma ou subtração de um AR com um MR são chamadas de operações *auto-modify*. As operações *auto-modify* podem ser utilizadas em operações de auto-incremento/decremento quando o valor a ser acionado/subtraído do AR é maior que o valor de *offset* implícito na AGU (geralmente +/- 1). A utilização de MRs em instruções de endereçamento indireto requer que estes registradores sejam carregados com *offsets* que depois serão utilizados, juntamente com AR, para o cálculo do endereço do próximo dado ($AR \pm MR$). Deste modo, a utilização de operações *auto-modify* implica em um custo adicional de carregamento destes registradores.

Capítulo 3

O problema de OA

Como já foi dito anteriormente, o problema de *Offset Assignment* (OA) consiste em encontrar uma disposição das variáveis locais e escalares alocadas na pilha a fim de minimizar o número de instruções aritméticas para cálculo de endereço, usando instruções com auto-incremento ou auto-decremento.

Nem sempre é possível utilizar instruções de auto-incremento e auto-decremento, pois dependendo do valor disponível no conjunto de instruções do processador para incremento ou decremento, algumas posições de memória não conseguem ser acessadas a partir do valor corrente do registrador de endereço. Por exemplo, suponha que o valor do incremento e do decremento seja 1, que as variáveis “a”, “b” e “c” estejam em seqüência na memória, e que somente exista um registrador de endereço (AR). Uma seqüência de acesso do tipo “acb” não consegue ser realizada sem a inserção de uma instrução explícita para atualizar o AR. Isto ocorre porque de início o AR será carregado com o endereço de “a” e a distância entre “a” e “c” é maior do que 1, que é o valor máximo usado para incremento ou decremento do AR em instruções do tipo auto-incremento ou auto-decremento. Portanto, para atualizar o AR, é preciso então inserir explicitamente uma instrução aritmética para somar 2 ao AR. Estas instruções são chamadas de *instruções de atualização*.

O problema de OA foi primeiramente estudado por Bartley [5] e por Liao et al [18]. Este último provou que o problema em grafos chamado de Cobertura por Caminhos de Peso Máximo, que é NP-Completo, pode ser reduzido ao problema OA quando possui somente um registrador de endereço e *offset* 1. Com isso, uma série de heurísticas foram criadas para resolver o problema (discutidas a seguir).

O problema de OA é chamado de *Simple Offset Assignment* (SOA) quando só existe um registrador de endereço (AR), e de *General Offset Assignment* (GOA) quando está disponível mais de um registrador de endereço.

3.1 Conceitos Básicos

Para facilitar o entendimento das soluções para o problema OA, serão introduzidos abaixo alguns conceitos utilizados neste âmbito.

Dada uma operação $z = x \text{ op } y$, define-se como *seqüência de acessos* a ordem em que as variáveis da operação são acessadas na memória (i.e. xyz). Logo, a seqüência de acessos para um conjunto ordenado de operações é simplesmente a concatenação das seqüências de acessos para cada operação na ordem apropriada. Como consequência, dado um bloco básico, há uma única seqüência de acessos para este bloco. Na Figura 3.1(b) é mostrada a seqüência de acessos relativa ao código da Figura 3.1(a).

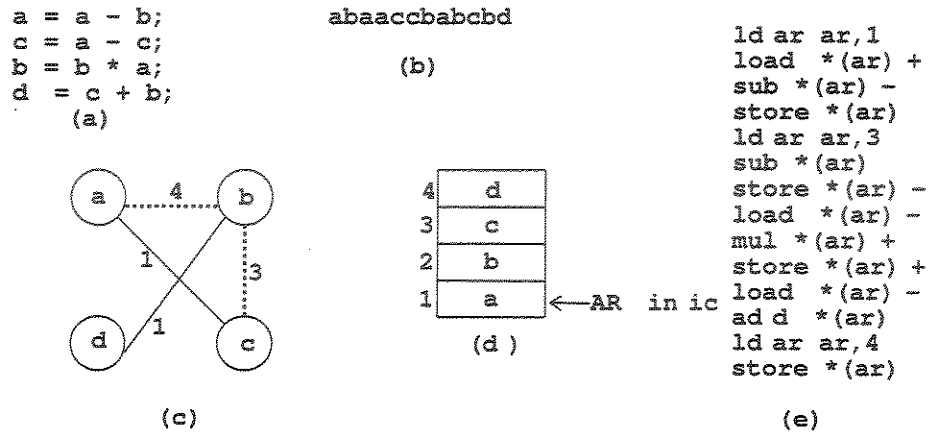


Figura 3.1: (a) Trecho de um programa em C (b) Seqüência de acessos (c) grafo de acessos (d) *offset assignment* (e) Código em *assembly* para o trecho do programa.

É possível resumir os padrões em que as variáveis são acessadas por um grafo não direcionado com pesos nas arestas. Este grafo é chamado de *grafo de acessos*. O grafo de acessos $G(V, E)$ é derivado da seqüência de acessos da seguinte maneira: cada vértice $v \in V$ no grafo corresponde a uma única variável, e uma aresta $e = (v_i, v_j) \in E$, entre os vértices v_i e v_j com peso $w(e)$, existe se as variáveis i e j forem adjacentes $w(e)$ vezes na seqüência de acessos, não importando a ordem de acesso. A Figura 3.1(c) mostra o grafo de acessos referente à seqüência de acessos da Figura 3.1(b).

Uma solução para o problema OA, ou seja, um *offset assignment*, é uma ordem em que as variáveis escalares estão dispostas na memória. Na Figura 3.1(d) é mostrada uma possível solução de OA para as variáveis do programa da Figura 3.1(a). Na Figura 3.1(e) é mostrado como fica o código *assembly*, se for utilizado a disposição das variáveis na memória mostrada na Figura 3.1(d). Neste caso supomos que: as operações são

todas feitas sobre o acumulador, só existe um AR, e o auto-incremento/decremento das instruções de endereçamento indireto é sempre 1.

Outro conceito importante para o entendimento de algumas soluções para o problema OA é o conceito de interferência entre duas variáveis. Duas variáveis a e b interferem quando estão vivas ao mesmo tempo em algum ponto do programa [19]. A partir das informações de interferência de cada variável, pode-se montar um grafo, chamado de grafo de interferência $G(V, E)$, onde V é um conjunto de vértices que representam as variáveis do programa, e E é um conjunto de arestas. Uma aresta (a, b) só está presente em E se as variáveis a e b interferem.

Outro sub-problema utilizado em OA é coloração de um grafo [28]. Uma coloração de k cores de um grafo $G = (V, A)$, não direcionado, é uma função $c : V \rightarrow \{1, 2, \dots, k\}$, tal que $c(u) \neq c(v)$ para toda aresta $(u, v) \in A$. Em outras palavras, os números $1, 2, \dots, k$ representam k cores, e vértices adjacentes têm que possuir cores diferentes. Obter uma coloração mínima para um grafo consiste em determinar o número mínimo de cores necessárias para colori-lo, dado que vértices adjacentes devem possuir cores distintas. Este problema é NP-Difícil [11], e, portanto, na prática, é resolvido por meio de heurísticas. Na resolução do problema OA, coloração de variáveis no grafo de interferência é utilizada com o objetivo de unir variáveis que não interferem, diminuindo assim o número de variáveis consideradas no problema. A heurística utilizada para resolver coloração é baseada no método de coloração de vértices de um grafo descrito por Kempe [28].

3.2 Simple Offset Assignment (SOA)

O problema OA, quando existe somente um registrador de endereço, é chamado de *Simple Offset Assignment* (SOA). SOA foi primeiramente estudado por Bartley [5] e Liao et al [18]. Liao et al [18] formulam OA como um problema de cobertura em grafos, chamado de Cobertura por Caminhos de Peso Máximo (MWPC¹), resolvendo este com uma heurística baseada no algoritmo de árvore geradora de Kruskal [12]. Para resolver SOA, Liao et al. [18] propuseram o *grafo de acessos* que, como já dito anteriormente, é composto de forma que cada vértice $v \in V$ no grafo corresponda a uma única variável. Uma aresta $e = (v_i, v_j) \in E$, com peso $w(e)$, indica que as variáveis i e j são adjacentes $w(e)$ vezes na seqüência de acessos. Portanto, no que diz respeito ao grafo de acessos, o custo de um *assignment* é igual à soma dos pesos de todas as arestas que conectam variáveis associadas a localizações de memória não adjacentes. A Figura 3.1(c) mostra o grafo de acessos (arestas pertencentes à cobertura estão pontilhadas) referente ao programa da Figura 3.1(a). A heurística utilizada em [18] é um algoritmo guloso que, a cada passo, escolhe

¹Do inglês *Maximum Weight Path Cover*

uma aresta para compor a cobertura por caminhos. Para que a cobertura permaneça um caminho, é necessário escolher uma aresta tal que o grau dos vértices do caminho não fique maior do que 2 e que não forme ciclo no caminho que está sendo composto. Estas condições precisam ser respeitadas visto que as variáveis em ordem na memória podem ser vistas como um caminho, pois uma variável pode ter no máximo dois vizinhos se não estiver na primeira posição ou na última posição da memória (nestes casos, só pode ter um vizinho). É escolhida a aresta que respeita as duas condições descritas acima, e que possui maior peso.

As Figuras 3.1(d)-(e) mostram a disposição das variáveis na memória obtida seguindo esta heurística e o código gerado para o programa supondo esta disposição. Primeiramente foram escolhidas as arestas (a, b) e (b, c) . Após, as arestas (a, c) e (d, b) foram rejeitadas. A primeira por formar ciclo e a segunda por deixar o vértice b com grau 3. Com isso, foi preciso inserir duas instruções para a atualização do AR. A instrução `ldar ar,3` representa a aresta (c, a) não coberta pelo caminho, e a instrução `ldar ar,4` representa a aresta (b, d) também não coberta pelo caminho.

Vários outros trabalhos foram feitos sobre SOA, e entre eles se destacam [16, 26, 25, 4, 29, 17].

Leupers e Marwedel em [16] aperfeiçoaram a heurística descrita em [18], criando um critério para escolher entre duas arestas quando elas possuem o mesmo peso. Este critério, chamado de *tie-break*, é definido da seguinte maneira: para um grafo de acesso $AG = (V, E, w)$, a função de *tie-break* $T : E \rightarrow N_0$ é definida por

$$T(e) = \sum_{e' \in E} w(e'), \quad e' \text{ e } e \text{ tendo pelo menos um dos vértices que as compõem em comum.}$$

Melhor explicando, a função *tie-break* de uma aresta e é a soma de todos os pesos das arestas que são compostas por pelo menos um dos vértices que compõem a aresta e .

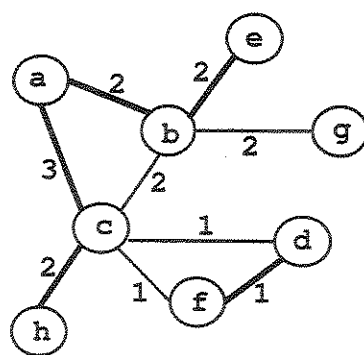
A aresta que possui o menor valor de *tie-break* é escolhida. Neste mesmo artigo, Leupers e Marwedel propõem também uma solução para SOA, quando há também MRs no processador. Esta solução executa um algoritmo muito semelhante ao algoritmo de Belady [6] para reposição de páginas em sistemas de memória virtual. Para o caso de SOA, o algoritmo executa na seqüência de operações da AGU computadas pelo algoritmo de SOA, ou melhor, na seqüência requerida de valores para os MRs.

Sudarsanam et al em [26] criaram um método para resolver o problema SOA quando o intervalo de auto-incremento/decremento pode ser de $-l$ a $+l$. Este problema é chamado de l-SOA. Para resolver este problema, Sudarsanam et al fazem uma adaptação do algoritmo de Liao et al [18], propondo uma nova função de custo para ser minimizada. Esta função de custo é calculada sobre o grafo de acessos completo, ou seja, incluindo as arestas

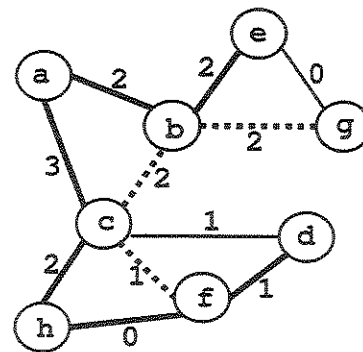
de peso zero. Ela computa o peso que cada aresta pode contribuir, somando os pesos de todas as arestas que podem tornar-se induzidas, se a aresta em questão for incluída na solução parcial. A aresta de maior contribuição é escolhida. Uma aresta é dita induzida, se esta aresta não pertence ao caminho, mas pertence a algum sub-grafo de $l + 1$ vértices, em que todos os vértices desse sub-grafo são vértices de algum sub-caminho do caminho de tamanho l . Escolhe-se a aresta com maior contribuição.

A Figura 3.2 mostra um exemplo de caminho retornado por este algoritmo. Na Figura 3.2(a) é dada uma seqüência de acessos, na figura 3.2(b) é mostrado o grafo de acessos referente a seqüência de acessos da Figura 3.2(a), com o caminho retornado pelo algoritmo de SOA de Liao et al [18] destacado. Na Figura 3.2(c) é mostrado o grafo de acessos, só que com o caminho escolhido pelo algoritmo de Sudarsanam et al em negrito. Nesta mesma Figura, observa-se que as arestas pontilhadas são as arestas induzidas pelo caminho, ou seja, arestas que não pertencem ao caminho, mas que não representam custo visto que o auto-incremento/decremento neste exemplo é 2. Os sub-grafos de $l + 1$ vértices usados para encontrar a solução do problema l-SOA são: *beg*, *abc*, *chf* e *dfh*. Nesta Figura, somente as arestas com peso zero que fazem parte do caminho são mostradas. Percebe-se que somente a aresta entre os vértices *c* e *d* representa um custo.

a b e b g b c d f c h c a b c a c
(a)



(b)



(c)

Figura 3.2: (a) seqüência de acessos (b) grafo de acessos com arestas pertencentes ao caminho retornado pelo algoritmo de Liao et al em negrito (c) grafo de acessos com arestas pertencentes ao caminho retornado pelo algoritmo de Sudarsanam et al em negrito, e as arestas induzidas em pontilhado.

Além da solução para o problema l-SOA, Sudarsanam et al propõem o uso de coloração de variáveis antes de aplicar o algoritmo para resolver SOA, com o objetivo de minimizar a quantidade de memória usada. Sudarsanam et al não reportaram os resultados obtidos quando aplica-se coloração antes do método para resolver SOA.

Rao e Pande em [25] propõem técnicas para otimizar a seqüência de acessos das variáveis aplicando transformações algébricas (comutatividade e associatividade) em árvores de expressão (e dentro de cada bloco básico). Foi feita uma nova formulação, considerando o problema da seqüência de acessos de custo mínimo. Baseadas nesta proposta, foram desenvolvidas novas heurísticas para determinar uma solução. Nesta proposta, diferentemente das existentes anteriormente, não é suposto que o escalonamento das instruções já tenha sido realizado. Neste artigo, são propostos dois algoritmos para resolver SOA. O primeiro algoritmo proposto transforma a seqüência de acessos de cada bloco básico, aplicando leis de comutatividade e associatividade nas expressões, a fim de que o grafo de acessos resultante tenha menos arestas, e que cada aresta possua pesos maiores, se comparado com o grafo de acessos da seqüência não otimizada. O segundo algoritmo proposto constrói o escalonamento das instruções e as seqüências de acessos incrementalmente, aplicando as leis de associatividade e comutatividade às expressões.

Atri et al em [4] propuseram um algoritmo incremental para resolver SOA. Este algoritmo tem como entrada uma seqüência de *offsets*, e tenta incluir no caminho arestas que não estavam previamente no caminho.

Outra colaboração relevante para o problema de SOA foi dada por Zhuang et al em [29]. Este algoritmo será explicado em detalhes, visto que ele está altamente relacionado com o algoritmo proposto nesta dissertação. O algoritmo proposto em [29] utiliza informação de vida das variáveis para tentar colocar na mesma posição de memória variáveis que não interferem entre si. Antes de unir dois vértices, sempre é verificado se eles não interferem. Além disso, após unir dois vértices, os grafos de acessos e interferência são atualizados a fim de refletirem esta união. Este algoritmo é separado em duas partes. Primeiramente, um conjunto composto por três regras para unir vértices é aplicado. Nestas regras, só são unidas duas variáveis se não houver aresta entre os dois vértices que as representam no grafo de interferência. Estas regras são:

- Regra 1: Unir todos os vértices que possuam grau zero no grafo de acessos com qualquer outro vértice;
- Regra 2: Unir, com seu vizinho, todos os vértices que possuam grau um no grafo de acessos;
- Regra 3: Unir, com o vizinho que possuir a aresta de maior peso, todos os vértices que possuam grau dois no grafo de acessos.

Após este passo, na segunda parte do algoritmo, em um processo iterativo, as variáveis são unidas incrementalmente. A cada passo, são escolhidas duas variáveis para serem unidas, baseado em uma estimativa de qual será o custo resultante de *offset assignment* após a união dessas variáveis. O algoritmo pára quando não houver mais variáveis para serem unidas.

Para escolher o par de vértices que será unido em cada passo do processo iterativo, é calculada, de forma aproximada, quantas instruções de endereçamento serão economizadas com a união dos dois vértices. Escolhe-se o par de vértices que possuir o maior valor resultante desta estimativa. A idéia básica para o cálculo deste valor, para dois vértices i e j , é usar o caminho resultante do algoritmo em [16] no grafo de acessos atual, ou seja, antes da união dos vértices i e j , para estimar quantas instruções seriam economizadas se os vértices i e j fossem unidos. Supondo que o vértice j será unido ao vértice i , o cálculo da economia de instruções de endereçamento é baseado nas seguintes afirmações:

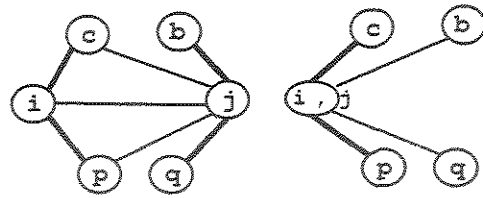
- O peso da aresta (i, j) , se ela ainda não pertencer ao caminho, é economizado;
- O peso de todas as arestas (k, j) , k vizinho de i no caminho são economizados. Na Figura 3.3(a), o peso das arestas (c, j) e (p, j) são economizados.
- O peso de todas as arestas (k, j) , k vizinho de j no caminho representarão novas instruções de endereçamento, e portanto aumentarão o custo da união. Na Figura 3.3(a) as arestas (b, j) e (q, j) aumentarão o custo da solução se i for unido a j .

O caso em que a aresta entre i e j já está no caminho é tratado separadamente. Neste caso, pode-se ver através da Figura 3.3(b) que os pesos das arestas (i, q) e (j, p) são economizados.

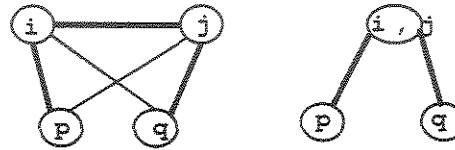
Se a união de dois ou mais pares de vértices resultar em o mesmo valor aproximado de economia de instruções de endereçamento, é aplicada a regra de desempate *tie-break* descrita em [16].

Outras observações sobre a segunda parte do algoritmo são que:

- O algoritmo primeiramente tenta unir os pares de vértices que são conectados no grafo de acessos, e só faz a união se a estimativa de custo economizado é maior do que zero. Só após tentar unir os pares de vértices que estão conectados no grafo de acessos é que se passa a unir os pares de vértices restantes. A razão para isto é que o custo resultante de SOA para grafos de acessos com poucos vértices de grau alto geralmente é menor do que para grafos com muitos vértices com grau alto. Devido a isto, tenta-se primeiro unir vértices que são vizinhos no grafo de acessos, visto que é menos provável que com este tipo de união o grau dos vértices no grafo de acessos



(a)



(b)

Figura 3.3: Casos possíveis a serem tratados pela heurística que calcula o número de instruções de endereçamento economizadas quando o vértice j é unido ao vértice i .

seja aumentado. Desta maneira, a união de vértices é dirigida via a propriedade de grau dos vértices.

- Com o objetivo de não “cair” em mínimos locais, no espaço de busca de uma solução, uma solução somente é guardada quando ela é melhor que a solução guardada anteriormente.
- Quando o algoritmo está tentando achar o par de vértices que são conectados no grafo de acessos cuja união pode resultar na maior economia de instruções de endereçamento, ele só considera aqueles em que a economia for um número positivo de instruções de endereçamento.

Li e Gupta em [17] propuseram três algoritmos para resolver o problema de SOA, todos baseados na idéia de colocar na mesma palavra de memória várias sub-palavras. O primeiro algoritmo proposto acha inicialmente um caminho no grafo de acessos, usando o algoritmo de Liao et al [18], e depois tenta agrupar as variáveis. O segundo algoritmo agrupa as variáveis, e depois acha um caminho, e o terceiro algoritmo faz o agrupamento à medida que vai escolhendo um caminho.

3.3 *General Offset Assignment (GOA)*

General Offset Assignment (GOA) é o nome dado ao problema de OA quando há mais de um registrador de endereço (AR). Este problema foi tratado primeiramente por Liao et al [18]. Eles fazem a generalização do SOA particionando o grafo de acessos em no máximo o número de registradores de endereço. Claramente, o problema GOA também é NP-Completo e, portanto, é preciso encontrar heurísticas para resolvê-lo.

A heurística utilizada em [18] baseia-se em construir as partições incrementalmente, uma partição para cada AR. Primeiramente, escolhe-se os dois vértices com maior penalidade, ou seja, com a maior soma de pesos de arestas incidentes, colocando-os numa mesma partição. Todos os outros vértices são colocados em uma outra partição. É aplicado SOA em cada partição, e a soma dos custos de cada partição acrescida do custo de inicialização do registrador de endereço de cada partição é comparado com o custo obtido pela aplicação de SOA sem particionamento. Caso o custo obtido com o particionamento seja menor, então tenta-se reparticionar a partição com os vértices de menores penalidades, prosseguindo com este procedimento até o número de partições ser igual ao número de ARs disponíveis ou o particionamento resultar num *assignment* de custo maior do que o feito na etapa anterior.

Na Figura 3.4 é mostrado um exemplo da resolução do problema de GOA para 2 registradores de endereço. A Figura 3.4(a) mostra uma seqüência de acessos e seu respectivo grafo de acessos (com as arestas escolhidas a partir do SOA pontilhadas). Já as Figuras 3.4(b)-(c) mostram o resultado do particionamento do grafo utilizando a heurística de [18], sendo as arestas pontilhadas as escolhidas. O grafo original foi particionado em dois. O grafo mostrado na Figura 3.4(b) ficou com os dois vértices de maior penalidade, e o mostrado na Figura 3.4(c) ficou com os vértices restantes. As novas subsequências para cada grafo originado do particionamento também são mostradas. É possível observar também, em pontilhado, os caminhos obtidos aplicando SOA em cada grafo de acessos. Na Figura 3.4(a), ou seja, no grafo de acessos não particionado, o caminho "abdce" foi escolhido, resultando num custo de alocação de 6. Portanto, 6 instruções explícitas de cálculo de endereço são necessárias para a solução encontrada. Na Figura 3.4(b), grafo de acessos de um dos grafos resultantes do particionamento, o caminho "cb" foi escolhido, tendo zero como custo de alocação. Na Figura 3.4(c), grafo de acessos do outro grafo resultante do particionamento, o caminho "ade" foi escolhido, tendo um custo de alocação de 1. Com isso, pode-se observar que o particionamento reduziu o custo do alocação de 6 para 2 (1 pela não cobertura da aresta (a, e) e mais 1 pela carga do outro AR, que sem o particionamento não era utilizado).

Leupers e Marwedel propuseram em [16] um algoritmo para resolver o problema de GOA que determina um tamanho variável para cada subconjunto que representará cada

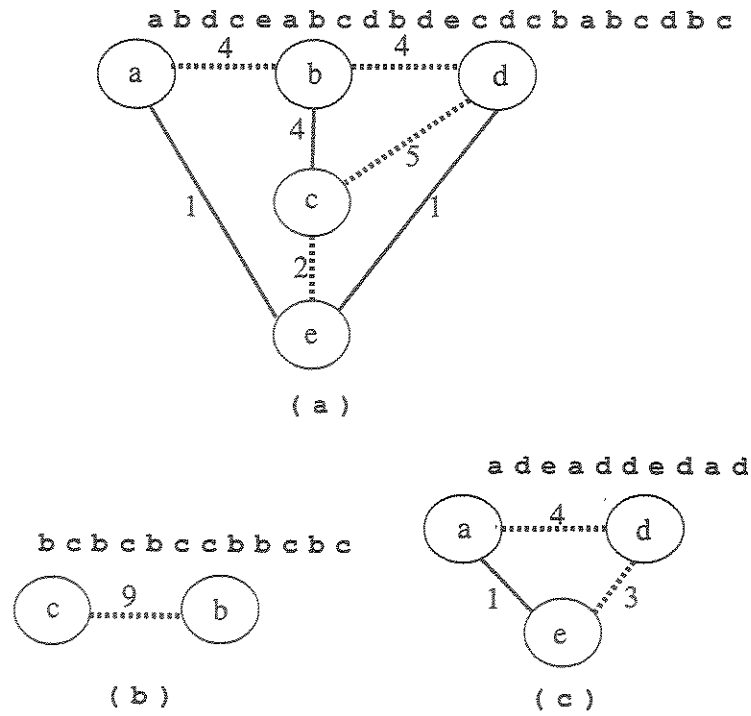


Figura 3.4: (a) Sequência de acessos e grafo de acessos (b) Subseqüência de acessos e grafo gerado por b, c (c) Subseqüência de acessos e grafo gerado por a, d, e.

AR, usando o algoritmo para resolver SOA [16] como uma subrotina que estimará os custos. O algoritmo começa preenchendo l ($l \leq k$, sendo k o número de registradores de endereço disponíveis) subconjuntos de variáveis com os l pares de vértices das l arestas disjuntas de maior peso do grafo de acessos. Cada par de vértices é colocado em um subconjunto. Se o grafo de acessos possui k ou mais arestas disjuntas, então $l = k$, caso contrário, todas as arestas são utilizadas, e $l \leq k$. Após esta fase, todas as variáveis v que sobraram são associadas ao V^* , sendo V^* o subconjunto em que o aumento no custo retornado pelo algoritmo de SOA causado pela adição de v à V^* é mínimo. Após todas as variáveis terem sido associadas a partições, a solução é obtida concatenando as l soluções de SOA de cada partição.

Uma outra heurística criada para o problema de GOA foi descrita em [15]. Esta técnica, de Leupers e David, resolve o problema de GOA utilizando otimização genética [10] (algoritmos genéticos). Nesta técnica, cada gene no cromossomo representa uma variável, e sua posição no cromossomo é o seu *offset*, visto que o cromossomo representa a solução para OA. Para achar uma solução, são feitas operações de mutação e de *crossover*.

Outra técnica para resolver o problema de GOA foi proposta por Zhuang et al em [29]. Esta técnica primeiramente tenta colorir as variáveis, aplicando o método de coloração de registradores descrito em [19]. Caso este método consiga usar no máximo $2K$ cores, sendo K o número de ARs disponíveis, então, no máximo duas cores serão colocadas por AR, e uma solução ótima foi encontrada. Caso contrário, uma heurística é utilizada para achar uma solução. Esta heurística é semelhante a utilizada por Leupers e David em [16]. Ela é composta por um laço, em que a cada iteração é colocada uma variável em uma das K partições. Este laço pára quando todas as variáveis já estão em alguma partição. Para escolher qual variável será colocada em qual partição, primeiramente é calculado o custo de adicionar cada variável a cada partição. O custo de selecionar uma variável v a uma partição p é retornado pelo algoritmo de SOA [29], executado para a partição p adicionada da variável v . Escolhe-se o par variável-partição que possui o menor custo. Caso haja mais de um par com o mesmo custo, duas regras são aplicadas para tentar acabar com o empate. Se ainda houver empate, escolhe-se um dos pares aleatoriamente. As regras são:

- Se a variável v é selecionada para a partição p , somam-se todas as arestas do grafo de acessos que vão de v para um vértice que já está em uma partição, exceto em p . Calcula-se este valor $w1$ para todos os pares que estão empatados, e mantém-se os pares que possuem $w1$ máximo;
- Se a variável v é selecionada para a partição p , $w2$ é o número de vizinhos de v no grafo de interferência que ainda não estão em uma partição. Calcula-se $w2$ para todos os pares que mantiveram-se empatados, escolhendo os que possuem o menor $w2$.

Após definidas as partições, executa-se o algoritmo de SOA de Zhuang et al [29] em cada uma das partições.

3.4 Alocação de Referências a Vetores

Outro problema relacionado com SOA é o problema de Alocação de Referências a Vetores (ARA)². A solução deste problema visa alocar registradores de endereço a acessos a vetores e matrizes dentro de um laço, de modo a maximizar a utilização de instruções de auto-incremento/decremento. ARA foi originalmente proposto e resolvido por Araújo et al [3], usando uma heurística baseada no algoritmo de emparelhamento em grafos bipartidos. Esta formulação englobava apenas laços formados por um único bloco básico, ou seja,

²Do inglês: *Array Reference Allocation*

sem instruções de desvio no seu corpo. Por este motivo, esta formulação foi denominada Alocação Local de Referências a Vetores (*Local Array Reference Allocation* – LARA).

Posteriormente, Leupers *et al* [24] propuseram uma solução exata para LARA, baseada na técnica de *branch-and-bound*, a qual utiliza a solução proposta em [2] como limitante inferior na sua solução.

Uma generalização de LARA para laços de programa contendo múltiplos blocos básicos é conhecida como o problema de Alocação Global de Referências a Vetores (*Global Array Reference Allocation* – GARA). A primeira solução para GARA foi proposta por Cintra e Araújo [8], e posteriormente aprimorada por Ottoni *et al* em [22, 21].

Capítulo 4

União de Variáveis no Problema de SOA (CSOA)

Coalescing-based Simple Offset Assignment (CSOA) é como foi chamada a heurística proposta nesta dissertação para resolver o problema de SOA. A idéia central desta heurística é coletar informações de vida das variáveis, construir o grafo de interferência a partir destas informações, e utilizá-lo para que variáveis que não estejam vivas ao mesmo tempo sejam alocadas na mesma posição de memória, caso isto diminua o número de instruções de endereçamento necessárias para atualizar o registrador de endereçamento. A técnica CSOA foi pensada de maneira a não unir variáveis indiscriminadamente, mas tentar tornar adjacentes na memória variáveis que possuam vários acessos consecutivos. Isto faz com que a distância entre variáveis que são acessadas consecutivamente seja diminuída. CSOA tenta unir variáveis objetivando uma diminuição no custo de *offset assignment* e uma diminuição na quantidade de memória usada. Para que este objetivo seja atingido, CSOA simultaneamente vai unindo variáveis e formando uma solução para o problema SOA. A cada passo do algoritmo, ou a aresta de peso máximo é escolhida para compor o caminho, ou une-se dois vértices, escolhendo-se realizar a operação que resulte em um menor número de instruções explícitas de endereçamento.

CSOA é uma extensão à maioria das heurísticas anteriores que resolvem SOA [18, 25, 4, 14]. O algoritmo CSOA é baseado no algoritmo de Liao et al [18] com a heurística de desempate de Leupers e Marwedel [16] para decidir entre arestas com o mesmo peso.

A heurística CSOA será explicada abaixo através do pseudo-código mostrado no Algoritmo 1.

Liao et al tentam formar um caminho máximo no grafo de acesso, ordenando as arestas do grafo de acessos em ordem decrescente de seus pesos. Após este passo, o algoritmo em [18] itera até que todos os vértices sejam inseridos no caminho, ou não haja mais arestas para escolher. Em cada passo da iteração, a aresta *válida* de maior peso é escolhida.

Algorithm 1 Coalescing-Based SOA

Entrada: seqüência de acessos L_{AS} ,
grafo de interferência $G_I(V_I, E_I)$.
Saída: offset assignment.

```
(1)  $G_A(V_A, E_A) \leftarrow \text{BuildAccessGraph}(L_{AS});$   
(2)  $L \leftarrow$  lista ordenada de  $E_A$ ;  
(3)  $coal \leftarrow$  falso;  
(4)  $sel \leftarrow$  falso;  
(5) repita  
(6)    $rebuild \leftarrow$  falso;  
(7)    $(coal, u, v, csave) \leftarrow \text{FindCandidatePair}(G_I, G_A);$   
(8)    $sel \leftarrow \text{FindEdgeValidNotSel}(L, e);$   
(9)   se  $(coal \ \&\& \ sel)$   
(10)     se  $(csave \geq w(e))$   
(11)        $rebuild \leftarrow$  verdadeiro;  
(12)     senão  
(13)       marque  $e$  como selecionada;  
(14)   senão  
(15)     se  $(coal)$   
(16)        $rebuild \leftarrow$  verdadeiro;  
(17)     senão se  $(sel)$   
(18)       marque  $e$  como selecionada;  
(19)   se  $(rebuild)$   
(20)      $\text{RebuildAccessGraph}(G_A, u, v);$   
(21)      $\text{RebuildInterferenceGraph}(G_I, u, v);$   
(22)      $\text{RebuildL}(L, G_A);$   
(23) até  $(!(coal \ || \ sel))$   
(24) retorne  $\text{BuildOffset}(G_A);$ 
```

Considere como *aresta válida* uma aresta não selecionada, ou seja, que ainda não esteja no caminho, cuja inclusão no caminho não cause ciclo e não aumente o grau de nenhum vértice para mais de dois. Já no Algoritmo 1, em cada passo da iteração, ao invés de sempre escolher uma aresta, uma nova alternativa é considerada: unir vértices. Especificamente, uma das duas operações é realizada:

1. Unir dois vértices u e v no grafo de acesso, se eles não interferem.
2. Selecionar a aresta válida de maior peso da lista de arestas ordenadas L , como na técnica de Liao et al.

No Algoritmo 1, a função *FindCandidatePair* tenta encontrar dois vértices candidatos para a união. Esta função retorna uma quádrupla $(coal, u, v, csave)$, onde *coal* é um sinalizador que é ativado se há dois vértices u e v que podem ser unidos, e *csave* é o número de instruções de endereçamento que serão economizadas caso u e v sejam unidos.

A fim de encontrar dois vértices para serem unidos, a função *FindCandidatePair* (linha (7)) procura entre todas as combinações de dois vértices u e v no grafo de interferência, aqueles que satisfazem as seguintes condições:

1. A aresta (u, v) não pertence ao grafo de interferência;
2. Unir u e v não cria ciclo no grafo de acessos, considerando somente as arestas selecionadas para compor o caminho;
3. Unir u e v não causa que o vértice resultante da união tenha grau maior do que dois no grafo de acessos, considerando somente as arestas selecionadas para compor o caminho.

Dentre todos os pares de vértices que satisfizerem as condições acima, o par u e v escolhido é aquele cuja sua união resultará no maior *csave*, ou seja, que a sua união resulta numa maior economia. A economia resultante é o número de instruções de atualização que serão economizadas devido à operação realizada.

Para calcular *csave*, a função *FindCandidatePair* computa as seguintes regras, onde $Adj_{sel}(y)$ é o conjunto de vértices adjacentes a y no grafo de acesso, considerando somente as arestas já selecionadas para compor o caminho:

- $\forall x \in (Adj_{sel}(u) - Adj_{sel}(v))$, adicione $w(x, v)$ ao *csave*; ($w(x, v)$ instruções de atualização serão economizadas). Isto significa que toda a aresta que seja entre v e um vértice x adjacente a u no caminho, e adjacente a v , mas não no caminho, é economizada.

- $\forall x \in (Adj_{sel}(v) - Adj_{sel}(u))$, adicione $w(x, u)$ ao *csave*; ($w(x, u)$ instruções de atualização serão economizadas).
- Adicione o peso da aresta entre u e v ao *csave*, se a aresta não foi selecionada para compor o caminho ainda; ($w(u, v)$ instruções de atualização serão economizadas caso a aresta (u, v) não esteja selecionada ainda).

Para melhor entendimento, considere a Figura 4.1. De acordo com as regras mostradas acima e a Figura 4.1, o valor de *csave*, quando u e v são unidos, é o peso da aresta (x, v) (uma vez que x é adjacente a u , a aresta (x, u) está selecionada, e a aresta (x, v) não está selecionada para fazer parte do caminho) mais o peso da aresta não selecionada (u, v) . O valor de *csave* torna-se 6, 4 da aresta (u, v) e 2 da aresta (x, v) .

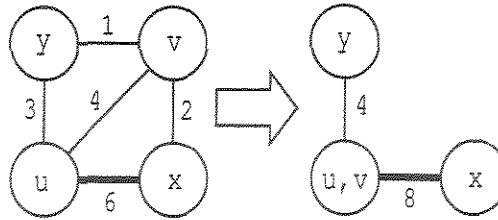


Figura 4.1: (a) Grafo de Acessos (b) Grafo de acessos depois da união das variáveis u e v .

Após este passo, na linha (8) do algoritmo 1, a função *FindEdgeValidNotSel* procura por uma aresta válida e com peso máximo $w(e)$ na lista ordenada de arestas L , e, se a aresta for encontrada, a variável sinalizadora *sel* torna-se verdadeira. Este passo é realizado para encontrar a aresta de peso máximo que ainda não faz parte do caminho. Após procurar pela aresta de maior peso, e por dois vértices que possam ser unidos, escolhe-se realizar a operação que resulte numa maior economia de instruções de atualização do AR.

Finalmente, se ambas as variáveis sinalizadoras *coal* e *sel* são verdadeiras (linha (9)), o Algoritmo 1 realiza a operação que resulta numa maior economia de instruções de atualização do AR. Se somente uma delas é verdadeira, realiza-se a operação que é sinalizada pela variável com valor verdadeiro, ou seja, seleciona aresta, se *sel* é verdadeira, e une vértices, se *coal* é verdadeira. Caso as duas variáveis *coal* e *sel* sejam falsas, não há mais operações a realizar, e o algoritmo termina.

Quando dois vértices u e v são unidos, partes do grafo de acessos e do grafo de interferência precisam ser reconstruídas para que estas estruturas reflitam esta operação. Isto é realizado nas linhas (19)-(22) do Algoritmo 1. No grafo de acessos, é necessário redirecionar todas as adjacências de u e v para o novo vértice (uv) resultante da união. No grafo de interferência, é preciso que o vértice (uv) , resultante da união, interfira com todos os vértices que eram adjacentes a u ou a v no grafo de interferência anterior.

A função *RebuildL*, linha (22) do Algoritmo 1, reconstrói a lista ordenada de arestas, isto é, L , a partir do novo grafo de acessos. O Algoritmo 1 acaba quando não há mais arestas válidas que possam ser escolhidas para fazer parte do caminho e não há mais vértices para serem unidos. Esta condição é testada pelas variáveis sinalizadoras *sel* e *coal* na linha (23) do Algoritmo 1.

4.1 Análise de Complexidade do Algoritmo CSOA

Nesta secção, é feita a análise de complexidade de tempo do algoritmo CSOA no pior caso. Nesta análise, considere m o comprimento da seqüência de acessos, e n o número de variáveis consideradas por CSOA. No Algoritmo 1, a complexidade da função *BuildAccessGraph* é $O(m + n^2)$. A operação de ordenação (linha (2)) tem complexidade $O(n^2 \log n)$. Depois disso, o laço *repita-até* pode ser executado no máximo $2(n - 1)$ vezes, ou seja, no máximo $n - 1$ arestas podem ser selecionadas, e no máximo $n - 1$ operações de união de vértices podem ser feitas. O laço *repita-até* é dominado pela função *RebuildL*, que possui complexidade $O(n^2 \log n)$. Logo ele possui complexidade $O(n^3 \log n)$. Finalmente, CSOA possui complexidade $O(m + n^3 \log n)$.

É importante ressaltar que esta análise é de pior caso, e que, na prática, este algoritmo mostrou-se bastante eficiente.

4.2 Exemplo de CSOA

Para melhor ilustrar CSOA, considere o fragmento de programa na Figura 4.2(a). Em cada ponto do programa é mostrado o conjunto de variáveis vivas (supondo que somente g esteja viva na saída do programa). Quando o Algoritmo 1 é aplicado a este exemplo, ele recebe como entrada o grafo de interferência, mostrado na Figura 4.2(b) e a seqüência de acessos (Figura 4.2(c)).

À medida que o algoritmo prossegue, ele produz a cada passo da iteração um grafo de acessos. Os grafos de acessos produzidos são mostrados na Figuras 4.2(d)-(j). Após isto, o algoritmo chega a um solução final. As arestas selecionadas durante a execução do algoritmo estão destacadas na Figura. A configuração de memória achada ao fim da execução do algoritmo é mostrada na Figura 4.2(k).

Embora não esteja ilustrado, o leitor deve lembrar-se de que, sempre que dois vértices do grafo de acessos são unidos, estes vértices são também unidos no grafo de interferência. Na primeira iteração (Figura 4.2(d)), a aresta (a, c) é selecionada, visto que não há um par de vértices cuja união produza uma economia de 2 ou mais instruções de atualização de registrador de endereço. Na próxima iteração, a melhor escolha é unir os vértices b e

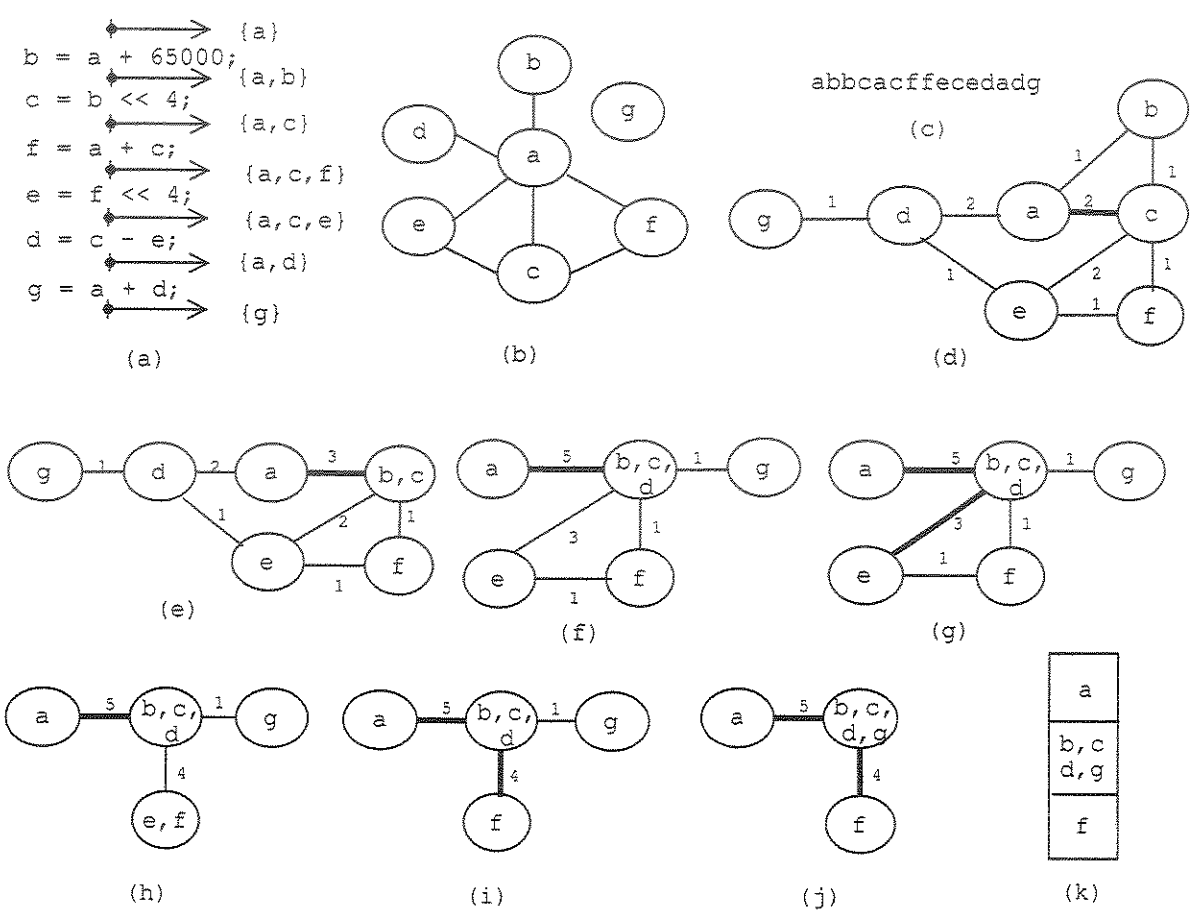


Figura 4.2: (a) Um fragmento de código em C, com informações de vida em cada ponto do programa. (b) O grafo de interferência das variáveis. (c) A seqüência de acesso deste fragmento de programa. (d)-(j) O grafo de acessos resultante depois de cada iteração do algoritmo. (k) O *layout* da memória. As arestas selecionadas estão em destaque.

c , dado que esta operação resulta numa economia de 2, correspondente as arestas (a, b) e (b, c) . O vértice resultante da união de b e c , (bc) , torna-se adjacente aos vértices que eram adjacentes a b ou a c no grafo de acesso resultante da iteração anterior, isto é, a , e e f . Note que o peso da aresta entre os vértices a e (bc) é 3, que é a soma dos pesos das arestas (a, b) e (a, c) no grafo de acessos resultante da iteração anterior. O algoritmo prossegue, escolhendo entre unir dois vértices ou selecionar uma aresta para o caminho, até que não seja mais possível realizar nenhuma dessas operações, resultando na Figura 4.2(j).

O custo final resultante da solução obtida pelo algoritmo CSOA para este exemplo é zero, visto que todas as arestas do grafo de acesso final são selecionadas. Para este exemplo, o algoritmo de Liao et al resulta em uma solução de custo quatro.

Capítulo 5

União de Variáveis no Problema de GOA (CGOA)

Coalescing-based General Offset Assignment (CGOA) é uma nova técnica criada para ser utilizada juntamente com a heurística CSOA, descrita no Capítulo 4. Com o objetivo de aumentar as oportunidades de união de variáveis pelo método CSOA, o particionamento feito por CGOA se baseia na informação de vida das variáveis, tentando colocar cada variável v na partição que possua as variáveis que menos interfiram com v . Este particionamento é guloso e, a cada passo, vai colocando uma das variáveis que ainda não pertence a nenhuma das partições naquela com a qual ela possui menos interferências. Esta técnica foi desenvolvida para melhorar o desempenho do CSOA que será aplicado em cada partição.

O algoritmo que descreve o particionamento utilizado nesta técnica está descrito no Algoritmo 2. Este algoritmo recebe como entrada a seqüência de acessos, o grafo de interferências e o número de registradores de endereço disponíveis. Como saída, ele retorna o conjunto de partições de variáveis. Inicialmente, como pode ser visto na linha (1) do Algoritmo 2, a estrutura de dados que conterá as partições é inicializada, baseando-se no número de ARs disponíveis. Após, na linha (2), é calculado o número de interferências que cada variável possui, utilizando para isto o grafo de interferência. Depois de calculado o número de interferências de cada variável, ordenam-se as variáveis, em ordem decrescente, utilizando como critério o número de interferências previamente calculado de cada uma, como é mostrado na linha (3) do Algoritmo 2. É nesta ordem que as variáveis vão ser colocadas nas partições. O trecho de código mostrado da linha (5) até a linha (10) escolhe a que partição a variável em questão será associada. Logo, para cada partição, calcula-se o número de interferências da variável em questão com todas as variáveis da partição, como é mostrado na linha (7) do algoritmo. Após este passo, coloca-se a variável na partição em que o número de interferências é menor, linha (11) do algoritmo. Depois de

todas as variáveis já terem sido associadas a alguma das partições, o algoritmo retorna o conjunto de partições. Um detalhe importante não mostrado no algoritmo, mas que a técnica utiliza, é que quando o número de interferências de uma variável com mais de uma partição é o mesmo, a variável é associada à partição que possui o menor número de variáveis, pois esta é uma maneira de ajudar no balanceamento das partições.

Algorithm 2 Algoritmo usado no particionamento do CGOA

Entrada: seqüência de acessos L_{AS} ,

grafo de interferência $G_I(V_I, E_I)$,

número de registradores de endereçamento $NARs$.

Saída: conjunto de subconjuntos disjuntos de variáveis $PartitionsSet$.

- (1) $PartitionsSet \leftarrow InicSetofPartitions(NARs)$;
 - (2) $IgDegree \leftarrow CalculateIgDegree(G_I(V_I, E_I))$;
 - (3) $VarsOrdByDegree \leftarrow FillVarsReverseOrdedByDegree(IgDegree)$;
 - (4) para todo $v \in V$ faça
 - (5) $MinNInt \leftarrow NVARs + 1$;
 - (6) para toda $Partition \in PartitionsSet$ faça
 - (7) $NInt \leftarrow CalculateNInterferences(VarsOrdByDegree, Partition)$;
 - (8) se $NInt < MinNInt$ então
 - (9) $MinNInt \leftarrow NInt$;
 - (10) $NPartMinNInt \leftarrow Partition$;
 - (11) PutVarInPartition($VarsOrdByDegree, NPartMinNInt, PartitionsSet$);
 - (12) retorne $PartitionsSet$;
-

Depois de realizado o particionamento das variáveis, é aplicado o algoritmo CSOA em cada uma das partições. A solução resultante para CGOA é a concatenação das soluções resultantes da aplicação de CSOA em cada uma das partições.

5.1 Análise de Complexidade de CGOA

Nesta secção é feita a análise de complexidade temporal do algoritmo CGOA no pior caso. Para esta análise, considere $nars$ o número de registradores de endereço disponíveis, m o comprimento da seqüência de acesso, e n o número de variáveis consideradas pelo algoritmo. Inicialmente, será feita a análise de complexidade do algoritmo 2 de particionamento utilizado pelo algoritmo CGOA. A função *InicSetofPartitions*, na linha (1) do Algoritmo 2, leva tempo $O(nars)$. A função *CalculateIGDegree*, na linha (2) do Algoritmo 2, precisa percorrer todo o grafo de interferência, para contar o número de interferências de cada variável e, com isso, sua complexidade temporal é $O(n^2)$. Na linha (3) do Algoritmo 2, a função *FillVarsReverseOrderedByDegree* possui complexidade $O(n \log n)$, uma vez que precisa ordenar as variáveis em ordem decrescente dos seus números de interferências. Por fim, o laço mostrado nas linhas (4)-(11) do Algoritmo 2, possui complexidade temporal $O(n(n + nars))$, visto que a complexidade do laço é dominada pela função *CalculateNInterferences* na linha (7), que possui complexidade $O(n)$. A função *PutVarInPartition*, linha (11) do algoritmo 2, possui tempo constante. Logo, o Algoritmo 2 possui complexidade $O(n(n + nars))$. Após o particionamento, o algoritmo CGOA executa o algoritmo CSOA para cada partição. Esta etapa do algoritmo possui complexidade $O(nars(m + n^3 \log n))$, pois executa $nars$ vezes o algoritmo CSOA, que possui complexidade $O(m + n^3 \log n)$. Por fim, o algoritmo CGOA possui complexidade $O(nars(m + n^3 \log n))$. É importante ressaltar que esta complexidade é de pior caso. Na prática, pode-se esperar uma execução bem rápida, pois, se as partições são mais ou menos balanceadas, o CGOA vai executar até mais rápido que o CSOA. Dado que o particionamento do CGOA divide o problema em problemas CSOA menores, a soma dos tempos para executar o CSOA em todas as partições será menor do que executá-lo para todas as variáveis juntas.

5.2 Exemplo de Particionamento em CGOA

Para ajudar no entendimento do particionamento usado no algoritmo CGOA, considere o grafo de interferência mostrado na Figura 5.1(a). A Figura 5.1(b) mostra as variáveis ordenadas em ordem decrescente dos seus números de interferências. Neste exemplo, são considerados 2 registradores de endereço. As Figuras 5.1(c)-(h) mostram como ficam os grafos de interferência das partições, à medida que elas vão sendo construídas. A Figura 5.1(i) mostra os grafos de interferência de cada partição após o término do particionamento. Para construir as partições, as variáveis são consideradas na ordem mostrada na Figura 5.1(b). Inicialmente, a variável 9 é colocada na partição P0, visto que o número de interferências da variável 9 é o mesmo nas duas partições (Figura 5.1(c)). A variável

6 é colocada na partição P1 (Figura 5.1(d)), pois ela possui uma interferência na partição P0, e nenhuma na partição P1. As variáveis vão sendo colocadas nesta ordem, na partição que menos interferem até que todas elas estejam em uma partição. Na Figura 5.1(h), a variável 5 é colocada na partição P1 e não em P0 porque P1 possui menos variáveis que P0, visto que a variável 5 possuía o mesmo número de interferências nas duas partições. A Figura 5.1(i) mostra o particionamento final obtido.

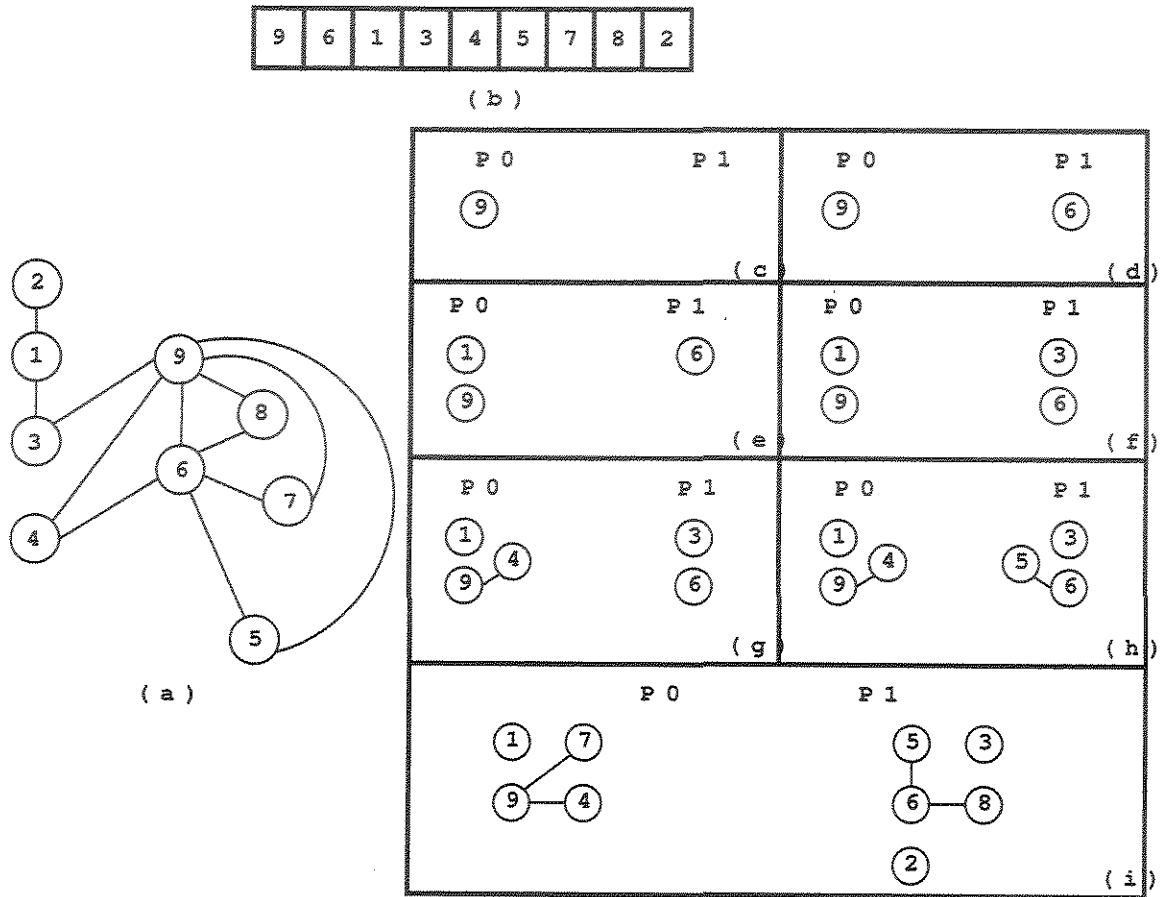


Figura 5.1: (a) Grafo de Interferência (b) Variáveis ordenadas em ordem decrescente dos seus números de interferência (c)-(h) Grafo de interferência das partições durante o particionamento (i) Grafos de interferência das partições após a execução do particionamento.

Capítulo 6

Resultados Experimentais

Visando medir a eficiência dos algoritmos descritos nos capítulos 4 e 5, estes foram implementados na ferramenta OffsetStone [14], criada por Rainer Leupers especialmente para o estudo de técnicas de OA. OffsetStone utiliza como *front-end* o compilador Lance [9], e extrai as seqüências de acessos de variáveis do programa. OffsetStone possui implementações para os seguintes métodos:

- SOA-OFU, uma heurística muito simples que associa *offsets* às variáveis de acordo com a ordem de seus primeiros usos no código;
- SOA-Liao, o algoritmo descrito em [18];
- SOA-TB [16], o algoritmo de [18] com um critério de desempate entre arestas de mesmo peso;
- SOA-GA [15], uma heurística baseada em algoritmos genéticos;
- SOA-INC-TB [14], uma combinação de duas heurísticas (SOA-incremental [4] e SOA-TB [16]);

Além das heurísticas propostas nos capítulos 4 e 5, foram implementadas também mais 4 heurísticas propostas anteriormente por outros pesquisadores. Estes métodos foram implementados com o objetivo de melhor comparar os resultados obtidos pelas principais soluções para os problemas de SOA e GOA, visto que todas estas técnicas não tinham sido diretamente comparadas. As quatro heurísticas são:

- Coal-SOA-Zhuang, o algoritmo de Zhuang et al, descrito em [29], que procura resolver o problema de SOA utilizando também informações de vida das variáveis [1];

- Color-SOA, o método descrito por Sudarsanam et al em [26], que faz uma coloração das variáveis para diminuir o número dessas, e com isso diminuir o tamanho da memória necessária, antes de executar a heurística de Liao et al [18];
- GOA-Leupers é uma heurística para resolver o problema de GOA de Leupers e Marwedel, descrita em [16];
- Coal-GOA-Zhuang, o algoritmo de Zhuang et al [29] que resolve o problema de GOA usando união de variáveis;

As instâncias usadas para avaliar as heurísticas foram extraídas dos programas que compõem o benchmark MediaBench [7]. As instâncias, ou seja, as seqüências de acessos, foram obtidas a partir do compilador Lance [9] da seguinte maneira:

- 1 Os códigos-fonte em C das aplicações são traduzidos para representação intermediária de código de três endereços (IR) pelo *frontend* do compilador LANCE, de modo a explicitar as seqüências de acessos;
- 2 A representação intermediária é otimizada com as seguintes técnicas [1]: eliminação de sub-expressões comuns, desdobramento de constantes, propagação de constantes, otimizações de salto, remoção de computação invariante do laço, eliminação de variáveis de indução, eliminação de sub-expressões globais comuns, eliminação de código morto, e propagação de cópia;
- 3 Da representação intermediária, a seqüência de acessos é extraída para cada bloco básico;
- 4 Uma vez que qualquer OA é válido dentro de uma função C inteira, um grafo de acesso global é construído por função, unindo os grafos de acessos locais dos blocos básicos que compõem a função;

Para obter os grafos de interferência, foi implementada uma análise de variáveis vivas [1], sobre a representação intermediária do LANCE após as otimizações de código. Esta análise calcula as variáveis vivas em cada ponto da representação intermediária, e a partir desta informação o grafo de interferência para cada função é montado. A análise de variáveis vivas precisou ser implementada, dado que esta análise não estava disponível no compilador LANCE, e que as técnicas anteriormente implementadas no *OffsetStone* não a utilizavam.

Na extração das seqüências de acessos, somente as variáveis que cabem em uma palavra de memória são consideradas. As variáveis que são ponteiros não são consideradas, haja vista que estas variáveis são geralmente alocadas em AR e não na pilha. É suposto que

Benchmarks	Liao	TB	GA	INC-TB	CSOA
adpcm	66,7%	63,8%	63,8%	63,8%	30,0%
epic	72,3%	70,0%	69,8%	69,8%	36,3%
g721	73,3%	70,5%	70,5%	70,5%	20,2%
gsm	72,2%	69,6%	69,6%	69,6%	13,4%
jpeg	68,8%	66,7%	66,5%	66,5%	21,9%
mpeg2	71,5%	69,6%	69,4%	69,5%	24,5%
pegwit	68,3%	62,2%	61,9%	61,9%	26,0%
pgp	70,8%	67,2%	67,1%	67,1%	22,5%
rasta	65,7%	64,7%	64,7%	64,7%	13,9%
Média	69,9%	67,1%	67,0%	67,0%	25,6%

Tabela 6.1: Custo de offset relativo ao custo obtido pela heurística SOA-OFU.

todas as variáveis que estão nas seqüências de acessos serão alocadas na pilha. Isto não é totalmente verdade, mas é uma boa aproximação, visto que os DSPs possuem geralmente poucos registradores de propósito geral.

6.1 Resultados Obtidos para SOA

Na Tabela 6.1 são mostrados os resultados obtidos pela heurística proposta no Capítulo 4, CSOA, e pelas cinco outras heurísticas que já estavam implementadas no *OffsetStone*. As percentagens mostradas para cada método na Tabela representam o custo de *offset* relativo ao custo obtido pela heurística SOA-OFU. SOA-OFU foi escolhida por ser a solução mais simples conhecido para SOA. Observe que, de acordo com a Tabela 6.1, a heurística CSOA reduz, em média, o número de instruções de atualização para 25,6% das inseridas pelo método SOA-OFU. Este resultado é muito superior aos obtidos pelas heurísticas existentes anteriormente, visto que o melhor algoritmo, SOA-GA, reduz o custo para 67%.

Foi implementado também, com o objetivo de comparar os métodos que utilizam união de variáveis, o algoritmo descrito em [26] por Sudarsanam et al. Este método inicialmente une as variáveis que não interferem, baseando-se no método de coloração de variáveis descrito por Kempe [28], e depois aplica o algoritmo de Liao et al [18] no novo conjunto de variáveis obtido após coloração. Este método foi implementado para ser comparado com CSOA. Os resultados obtidos por este método e pelo método Coal-SOA-Zhuang de Zhuang et al descrito em [29], considerando-se como métrica o custo de OA relativo ao custo obtido pela heurística SOA-OFU, são mostrados na Tabela 6.2. A partir da Tabela, nota-se que o algoritmo CSOA é superior aos outros algoritmos descritos, visto que ele

Benchmarks	CSOA	Color-SOA	Coal-SOA-Zhuang
adpcm	30,0%	37,2%	31,9%
epic	36,3%	53,7%	39,9%
g721	20,2%	37,1%	22,1%
gsm	13,4%	19,2%	14,6%
jpeg	21,9%	36,2%	26,6%
mpeg2	24,5%	43,1%	26,8%
pegwit	26,0%	51,3%	30,3%
pgp	22,5%	39,0%	26,3%
rasta	13,9%	21,8%	24,7%
Média	25,6%	35,8%	30,3%

Tabela 6.2: Custo de offset relativo ao custo obtido pela heurística SOA-OFU.

reduz, em média, o número de instruções de atualização para 25,6% . Este resultado é superior ao obtido pelo método Coal-SOA-Zhuang que reduz o número de instruções de atualização para 30,3%, e ao obtido pelo método SOA-Color que reduz para 35,8%.

Um outro resultado importante é o quanto de memória (pilha) é utilizado pelos *offsets* obtidos pelos métodos que fazem união de variáveis, comparando com os métodos que não o fazem. Este resultado pode ser observado na Tabela 6.3. As percentagens mostradas na Tabela 6.3 representam o tamanho da pilha resultante dos métodos baseados em união de variáveis relativo ao tamanho da pilha resultante dos métodos que não a utilizam. CSOA reduz o tamanho da pilha usada para armazenar variáveis locais para 28,6%, quando comparado com os outros cinco métodos [18, 16, 15, 14] implementados previamente no *OffsetStone*. O método de Color-SOA reduz o tamanho da pilha para 11,6%, e o método de Coal-SOA-Zhuang para 24%. É importante salientar, que embora CSOA seja o método que faz menos uniões entre variáveis, o custo de OA produzido por ele é menor do que os custos obtidos pelos outros métodos de SOA implementados nesta dissertação. Além disto, ao obter um *offset* melhor, CSOA requer menos memória que os demais algoritmos para armazenar o código resultante, além do ganho de desempenho.

A Tabela 6.4 mostra o número de variáveis temporárias em cada programa. Uma variável é considerada temporária se sua vida limita-se a somente um bloco básico. Esta informação é considerada importante, pois um grande número de variáveis temporárias aumenta a chance de realizar união de variáveis. A Tabela 6.4 mostra que, em média, 64,1% das variáveis produzidas pelo compilador LANCE são temporárias.

Benchmarks	CSOA	SOA-Color	Coal-SOA-Zhuang
adpcm	27,3%	15,7%	28,3%
epic	27,0%	11,6%	26,6%
g721	25,0%	13,3%	22,7%
gsm	21,5%	7,0%	19,8%
jpeg	34,7%	14,6%	25,7%
mpeg2	31,8%	12,2%	21,9%
pegwit	35,3%	9,7%	26,8%
pgp	31,5%	12,8%	24,7%
rasta	26,1%	9,9%	21,4%
Média	28,6%	11,6%	24,0%

Tabela 6.3: Tamanho da pilha relativo ao tamanho da pilha utilizada pelos métodos que não fazem união de variáveis.

Benchmarks	%Temporárias
adpcm	59,6%
epic	48,1%
g721	80,7%
gsm	86,6%
jpeg	65,2%
mpeg2	65,6%
pegwit	72,1%
pgp	67,5%
rasta	43,6%
Média	64,1%

Tabela 6.4: Percentagem de variáveis temporárias.

Benchmarks	CGOA	CGOA-Liao	GOA-Leupers
adpcm	7,3%	44,4%	23,2%
epic	22,8%	51,3%	30,3%
g721	6,8%	47,1%	27,3%
gsm	3,0%	32,4%	36,0%
jpeg	12,5%	46,6%	23,5%
mpeg2	15,2%	52,0%	28,9%
pegwit	10,8%	41,8%	19,1%
pgp	11,4%	44,9%	25,2%
rasta	10,4%	54,1%	12,0%
Média	11,3%	52,8%	27,9%

Tabela 6.5: Custo de offset relativo ao custo obtido pela heurística SOA-OFU, utilizando 2 ARs.

6.2 Resultados Obtidos para GOA

A Tabela 6.5 mostra os resultados obtido pelos métodos:

- CGOA, método proposto nesta dissertação e descrito no Capítulo 5;
- CGOA-Liao;
- O método criado por Leupers e Marwedel, GOA-Leupers, descrito em [16].

O método CGOA-Liao consiste no particionamento descrito no Capítulo 5 seguido da heurística de Liao et al descrita em [18]. Para obter os resultados mostrados na Tabela 6.5, são utilizados somente dois registradores de endereço. Os resultados são mostrados em termos do custo de *offset* relativo ao custo obtido pela heurística SOA-OFU, que atribui as variáveis a posições de memória na ordem em que elas aparecem no código. Pode-se observar que na média CGOA, para apenas 2 AR's, reduz para 11,3% o custo de *offset*. Os métodos de CGOA-Liao e GOA-Leupers reduzem o custo de *offset* para, respectivamente, 52,8% e 27,9%.

A Tabela 6.6 mostra os resultados obtido pelos métodos CGOA, CGOA-Liao e GOA-Leupers, quando quatro registradores de endereço são considerados. As percentagens mostradas são custo de *offset* de cada método comparados com o custo resultante do método SOA-OFU. A heurística CGOA reduz o custo de *offset* para 8,4%, a heurística CGOA-Liao para 36,4% e a técnica de Leupers e Marwedel para 14,6%. Novamente, a técnica CGOA obtém um melhor resultado.

Benchmarks	CGOA	CGOA-Liao	GOA-Leupers
adpcm	4,4%	25,1%	8,2%
epic	10,5%	31,5%	8,7%
g721	7,1%	30,6%	14,6%
gsm	4,4%	27,0%	36,7%
jpeg	9,2%	34,5%	15,2%
mpeg2	10,1%	38,2%	13,7%
pegwit	5,7%	24,2%	12,9%
pgp	7,8%	31,4%	14,3%
rasta	9,2%	46,2%	5,4%
Média	8,4%	36,4%	14,6%

Tabela 6.6: Custo de offset relativo ao custo obtido pela heurística SOA-OFU, utilizando 4 ARs.

A Tabela 6.7 mostra os resultados obtido pelos métodos CGOA, CGOA-Liao e GOA-Leupers, quando oito registradores de endereço são considerados. Como nas tabelas anteriores, as percentagens mostradas são o custo de *offset* de cada método comparadas com o gerado pelo método SOA-OFU. Os resultados obtidos para oito registradores de endereço pelos três métodos foram piores que os obtidos para quatro registradores. Provavelmente, o custo para atualizar inicialmente mais quatro registradores não compensa. No método CGOA, muitas partições acabam diminuindo as oportunidades para unir variáveis. A heurística CGOA reduz o custo de *offset* para 10,2%, a heurística CGOA-Liao para 37,2% e a técnica de Leupers e Marwedel para 22%.

A Tabela 6.8 mostra para quanto o método de CGOA diminuiu o tamanho da pilha de memória, quando utilizados dois, quatro e oito registradores de endereço. Os resultados são mostrados em percentagem, usando como base o tamanho de memória resultante do método SOA-OFU. A heurística reduz o tamanho da pilha para 35,5%, 46,7% e 58,2% respectivamente. Estes números comprovam que quanto mais partições menos oportunidades de unir de variáveis existe, visto que menos variáveis são atribuídas a cada partição. Obviamente, existe um número ideal de partições, que pode estar entre três e oito para o benchmark Mediabench.

As Tabelas 6.9, 6.10 e 6.11 fazem uma comparação entre os resultados obtidos por CGOA e por Coal-GOA-Zhuang, método criado por Zhuang et al e descrito em [29], usando respectivamente 2, 4 e 8 registradores de endereço. Devido a alta complexidade do método de Zhuang, e o grande número de variáveis existente em algumas instâncias dos programas do Mediabench, só foi possível executar o método de Zhuang para instâncias

Benchmarks	CGOA	CGOA-Liao	GOA-Leupers
adpcm	10,1%	23,7%	16,9%
epic	5,5%	19,9%	5,8%
g721	13,1%	29,7%	27,3%
gsm	7,1%	45,3%	55,7%
jpeg	9,3%	38,8%	27,8%
mpeg2	9,2%	34,5%	17,9%
pegwit	9,4%	31,5%	26,6%
pgp	8,2%	34,4%	24,0%
rasta	9,4%	39,7%	6,8%
Média	10,2%	37,2%	22,0%

Tabela 6.7: Custo de offset relativo ao custo obtido pela heurística SOA-OFU, utilizando 8 ARs.

Benchmarks	2ARs	4ARS	8ARS
adpcm	28,8%	44,5%	55,6%
epic	32,0%	37,3%	48,3%
g721	30,6%	42,7%	57,3%
gsm	27,3%	36,7%	46,2%
jpeg	43,0%	55,4%	68,5%
mpeg2	41,8%	52,3%	62,5%
pegwit	42,5%	54,5%	65,7%
pgp	53,2%	53,4%	68,5%
rasta	23,1%	47,8%	56,1%
Média	35,5%	46,7%	58,2%

Tabela 6.8: Percentagem de memória resultante do método CGOA relativo ao tamanho de memória usado pelos métodos que não utilizam união de variáveis.

Benchmarks	CGOA	Coal-GOA-Zhuang
adpcm	7,3%	8,7%
epic	15,5%	12,4%
g721	6,8%	8,5%
gsm	3,4%	9,4%
jpeg	11,6%	12,7%
mpeg2	13,9%	10,7%
pegwit	9,7%	10,3%
pgp	10,7%	10,8%
rasta	12,8%	12,3%
Média	6,8%	7,5%

Tabela 6.9: Custo de offset relativo ao custo obtido pela heurística SOA-OFU, considerando 2 ARs.

com menos de 270 variáveis. Com o objetivo de comparar os dois métodos, foi limitado também o número de variáveis de cada instância para o método CGOA.

Os resultados na Tabela 6.9 mostram que CGOA reduz o custo de *offset* para 6,8% e Coal-GOA-Zhuang para 7,5%, considerando como base o custo resultante do método SOA-OFU. Vale salientar que a heurística CGOA possui várias vantagens se comparada a heurística Coal-GOA-Zhuang, entre elas: é mais fácil para implementar, possui complexidade menor, e resulta em um menor custo de *offset*.

A Tabela 6.10 mostra os resultados obtidos por CGOA e Coal-GOA-Zhuang, quando 4 ARs estão disponíveis. CGOA reduz o custo de *offset* para 7,3% e Coal-GOA-Zhuang para 18,6%, considerando como base o custo resultante do método SOA-OFU. As soluções obtidas tanto por CGOA quanto por Coal-GOA-Zhuang, considerando 4 ARs são inferiores as obtidas considerando somente 2 ARs. Os resultados mostrados na Tabela 6.11 são mais inferiores ainda, visto que foram considerados 8 registradores de endereço.

Benchmarks	CGOA	Coal-GOA-Zhuang
adpcm	4,3%	12,1%
epic	8,6%	17,7%
g721	8,2%	17,5%
gsm	5,4%	25,4%
jpeg	8,1 %	20,4%
mpeg2	8,9 %	17,0%
pegwit	8,0%	22,1%
pgp	7,1 %	18,7%
rasta	8,3 %	19,4%
Média	7,3%	18,6%

Tabela 6.10: Custo de offset relativo ao custo obtido pela heurística SOA-OFU, considerando 4 ARs.

Benchmarks	CGOA	Coal-GOA-Zhuang
adpcm	10,1%	16,9%
epic	9,6%	26,0%
g721	14,8%	32,1%
gsm	9,0%	57,6%
jpeg	9,0 %	33,9%
mpeg2	9,3 %	27,1%
pegwit	14,5%	43,7%
pgp	8,6 %	30,1%
rasta	9,5 %	30,1%
Média	10,3%	31,4%

Tabela 6.11: Custo de offset relativo ao custo obtido pela heurística SOA-OFU, considerando 8 ARs.

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho, foi estudado o problema de *Offset Assignment* (OA) para DSPs. Este problema vem sendo estudado amplamente nos últimos anos, visto que os DSPs são arquiteturas muito irregulares, com poucos registradores de propósito geral, e modos de endereçamento restritos. De uma forma geral, DSPs não possuem modo de endereçamento indexado, empregando endereçamento indireto com aritmética de auto-incremento e auto-decremento, usando registradores de endereço para apontar para seus endereços de memória.

O problema de OA visa encontrar uma disposição das variáveis locais na memória de forma a minimizar o uso de instruções aritméticas para cálculo de endereço, usando instruções com auto-incremento ou auto-decremento, obtendo assim um código de melhor qualidade. A escolha de uma boa disposição das variáveis na memória acarreta uma diminuição no número de instruções geradas, e com isso, uma melhora no desempenho.

O problema OA é chamado de *Simple Offset Assignment* (SOA) quando há somente um registrador de endereço. Quando há mais de um registrador de endereço, ele é chamado de *General Offset Assignment* (GOA).

Nesta dissertação, foi feita uma extensa revisão bibliográfica, e foram propostas duas técnicas para a resolução de OA que utilizam informações de vida das variáveis para obter uma melhor disposição das variáveis na memória. A primeira resolve o problema de SOA, e foi chamada de *Coalescing-based SOA* (CSOA). A segunda, resolve GOA, e foi chamada de *Coalescing-based GOA* (CGOA). Além disso, foram implementadas as técnicas propostas, e mais 4 heurísticas descritas na literatura. Isto possibilitou, pela primeira vez, que os principais métodos que tratam o problema OA fossem comparados. Pode-se perceber, através dos resultados experimentais obtidos, que as soluções baseadas nas seqüências de acessos e no grafo de interferências, produzem códigos bem mais eficientes, considerando o tamanho do código gerado, do que as soluções baseadas somente nas seqüências de acessos.

Como trabalhos futuros, pode-se citar:

- Primeiramente, tentar acoplar o escalonamento de instruções no algoritmo de CSOA, tomando cuidado para que o escalonamento não modifique o grafo de interferência em que CSOA se baseia;
- Uma outra sugestão é adaptar a solução descrita por Sudarsanam et al [26], que resolve o problema de SOA, quando o auto-incremento/decremento pode ser maior que 1, a heurística CSOA;
- Implementar o algoritmo de Belady [6] para testar se a utilização de *modify registers* melhorará significativamente os resultados obtidos por CSOA;
- Colocar regras para desempate, quando mais de um par de vértices quando unidos resultam em uma mesma economia de instruções de *update* no algoritmo de CSOA. No algoritmo descrito no Capítulo 4, quando há empate, escolhe-se o primeiro par de vértices considerado;
- Testar o particionamento para o problema de GOA descrito por Zhuang et al em [29] com o algoritmo CSOA;
- Por fim, tentar desenvolver uma heurística para particionamento para GOA que use não só as informações de vida das variáveis, mas também as informações de acessos.

Referências Bibliográficas

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.
- [2] Guido Araújo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University, 1997.
- [3] G. Araujo, A. Sudarsanam, and S. Malik. Instruction set design and optimizations for address computation in DSP architectures. In *Proc. of the 9th. ACM/IEEE International Symposium on System Synthesis*, pages 102 – 107, November 1996.
- [4] Sunil Atri, J. Ramanujam, and Mahmut Kandemir. Improving offset assignment for embedded processors. *Lecture Notes in Computer Science*, 2017, 2001.
- [5] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software - Practice and Experience*, 22(2):101–110, 1992.
- [6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM System Journal*, 5(2):78–101, 1966.
- [7] M. Potkonjak C. Lee and W. Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture (Micro 30)*, December 1997.
- [8] M. Cintra and G. Araujo. Array reference allocation using ssaform and live range growth. In *Proc. of the ACM SIGPLAN 2000 LCTES*, pages 26–33, June 2000.
- [9] LANCE Retargetable C compiler. <http://ls12-www.cs.uni-dortmund.de/lance/>.
- [10] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
- [11] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, New York, 1979.

- [12] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
- [13] R. Leupers. *Code Optimization for Embedded Processors*. Kluwer Academic Publishers, 2000.
- [14] Rainer Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proceedings of the 12th International Conference on Compiler Construction*, April 2003.
- [15] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In *Proc. of the International Symposium on System Synthesis (ISSS)*, pages 3–8, 1998.
- [16] Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *International Conference on Computer-Aided Design (ICCAD)*, pages 109–112, 1996.
- [17] B. Li and R. Gupta. Simple offset assignment in presence of subword data. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'03)*, pages 12–23. ACM Press, October 2003.
- [18] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.
- [19] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [20] Desirée Ottoni, Guilherme Ottoni, Guido Araújo, and Rainer Leupers. Offset assignment through simultaneous variable coalescing. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES'03)*, volume LNCS 2826, pages 285–297. Springer Verlag, September 2003.
- [21] Guilherme Ottoni and Guido Araújo. Address register allocation for arrays in loops of embedded programs. *Microelectronics Journal*, 34(11):1009–1018, October 2003.
- [22] Guilherme Ottoni, Sandro Rigo, Guido Araujo, Subramanian Rajagopalan, and Sharad Malik. Optimal live range merge for address register allocation in embedded programs. In *Proceedings of the 10th International Conference on Compiler Construction, CC2001, LNCS 2027*, pages 274–288. Springer, April 2001.

- [23] P. Paulin, C. Liem, M. Cordeiro, F. Naçabal, and G. Goossens. Embedded software in real-time signal processing systems: Application and architecture trends. *Proceedings of the IEEE*, 85(3), March 1997.
- [24] A. Basu R. Leupers and P. Marwedel. Optimized array index computation in DSP programs. In *Proc. of the Asia South Pacific Design Automation Conference (ASP-DAC)*. IEEE, February 1998.
- [25] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 128–138, May 1999.
- [26] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Design Automation Conference*, pages 287–292, 1997.
- [27] S. Udayanarayanan and C. Chakrabarti. Address code generation for digital signal processors. In *Proceedings of ACM/IEEE Design Automation Conference (DAC'01)*, pages 155 – 164, June 2001.
- [28] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, second edition, 2001.
- [29] X. Zhuang, C. Lau, and S. Pande. Storage assignment optimizations through variable coalescence for embedded processors. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Processors (LCTES'03)*, pages 220–231. ACM Press, June 2003.