UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Análise da Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos e Combinação de Registradores

Luciana Leal Ambrosio

Orientadora: Mariza Andrade da Silva Bigonha
Co-orientador: Roberto da Silva Bigonha

24 de Maio de 2004

Conteúdo

Resumo

Alocação de registradores é a fase de compilação que decide quais valores do programa devem estar armazenados nos registradores físicos em cada ponto de execução do programa. Geralmente, existem poucos registradores na máquina, menos do que o necessário, fazendo com que alguns valores contidos em registradores sejam derramados para a memória. Quanto mais instruções de acesso à memória existirem no código, pior é a sua eficiência - em relação a seu tempo de execução. Uma alocação de registradores ótima reduz o número deste tipo de instruções. Porém, como este é um problema NP-completo, tenta-se resolvê-lo por meio de heurísticas, dentre as quais a mais utilizada é a coloração de grafos, muito embora ela não produza código eficiente em algumas situações. Este trabalho se propõe a implementar uma nova heurística para alocação de registradores baseada no algoritmo de Crescimento de Domínios Ativos de Ottoni e Araújo. Nesta implementação, os candidatos à alocação são denominados domínios ativos. Estes são unidos a cada iteração do algoritmo, até que o seu número total atinja o número de registradores disponíveis para alocação. Para se decidir qual par de domínios ativos será unido a cada iteração, todas as combinações de pares são avaliadas e a que resultar na inserção de um menor número de instruções de acesso à memória no código resultante é a escolhida para a união. A implementação deste algoritmo foi avaliada e comparada à implementação do algoritmo de George e Appel.

Capítulo 1

Introdução

Registradores são memórias pequenas, caras e rápidas que existem dentro da CPU, e que guardam valores freqüentemente utilizados durante a execução do programa. A alocação de registradores é o passo da compilação que determina quais valores: variáveis, temporários e elementos individuais de um vetor, devem estar em registradores em cada ponto da execução do programa, minimizando o tráfego entre os registradores da CPU e a memória. Alocação de registradores é importante porque registradores são recursos escassos e também porque em arquiteturas RISC, todas as operações, exceto *load* e *store*, são efetuadas no conteúdo de registradores, e não em memória. Tanto nas arquiteturas RISC como CISC, as operações de registradores para registradores são bem mais rápidas que as de registrador para memória ou de memória para memória. Já a Atribuição de Registradores *(Register Assignment)* é a fase da compilação que determina, para cada valor alocado, o registrador correspondente à alocação. Esta dissertação se concentra no passo de alocação de registradores.

Geralmente, por existirem poucos registradores na máquina, menos do que o necessário, alguns valores contidos em registradores são derramados para a memória. Derramar um valor para a memória significa que pelo menos um acesso à memória é feito para obtenção deste valor e isto aumenta o tempo de execução do programa, pois instruções de acesso à memória têm um tempo de execução maior que as de acesso a registradores. O problema de se encontrar uma alocação de registradores ótima é um problema NP-completo [18], logo deve-se buscar heurísticas para resolvê-lo.

Em um compilador que faz otimização global, o passo de alocação de registradores é quase sempre feito após a geração de código de baixo nível ou código de máquina. A alocação pode ser precedida por escalonamento de instruções [18] e por *software pipelining* [18], e pode ser sucedida por outro passo de escalonamento de instruções.

Os métodos mais antigos de alocação de registradores são abordagens locais, como a desenvolvida por Freiburghouse [13] e a utilizada no compilador PDP-11 BLISS [18] e seus descendentes na década de 70. A abordagem de Freiburghouse é hierárquica, pois partindo-se do princípio de que a maioria dos programas gasta a maior parte do seu tempo de execução executando laços, ela coloca mais peso nos laços mais internos *(inner loops)* do que nos mais externos *(outer loops)* e do que em código não contido em laços. A idéia é determinar, heuristicamente, os benefícios

de alocação de várias quantidades alocáveis. Estes valores são geralmente estimados multiplicando os benefícios obtidos para se alocar uma variável a um registrador por um fator baseado na profundidade do aninhamento do laço. Além disso, o fato de uma variável estar viva na saída ou entrada de um bloco básico também deve ser considerado. Um bloco básico é definido como uma sequência de zero ou mais instruções, onde nenhuma delas é uma instrução de desvio de controle [25]. As seguintes quantidades são definidas:

- *ldcost*: é o custo no tempo de execução de se fazer uma instrução de *load*;
- *stcost*: é o custo de uma instrução de *store*;
- *mvcost*: é o custo de uma instrução que move de registrador para registrador;
- *usesave*: é a economia para cada uso de uma variável que reside em um registrador em relação ao uso de uma que reside na memória;
- *defsave*: é a economia de cada definição de uma variável que reside em um registrador em relação a uma que reside na memória.

Logo, a economia no tempo de execução para uma variável v cada vez que o bloco básico B_i é executado é dada por:

$$netsave(v,i) = u * usesave + d * defsave - l * ldcost - s * stcost$$

onde u e d dão os números de usos e de definições da variável v no bloco i, e l é igual a 1 se um *load* de v for necessário no início do bloco B_i, e 0 caso contrário. Já s é igual a 1 se um *store* de v no final de B_i for necessário e 0 caso contrário.

Para se calcular a economia de tempo de um programa todo, faz-se:

$$10^d * (\sum_{i \in blocks(L)} netsave(v,i))$$

onde d é a profundidade do laço, L é um laço e i representa todos os blocos do programa. Esta equação fornece o benefício de se alocar v a um registrador no laço. Dado que existem R registradores disponíveis para alocação, depois de se computar as estimativas citadas acima, os R objetos com a maior estimativa são alocados aos registradores. A alocação para laços externos é feita de forma análoga.

Esta abordagem é simples de se implementar e é eficiente. Ela foi utilizada nas séries IBM 360 e 370, até que os métodos globais de alocação de registradores fossem possíveis.

A segunda abordagem local, utilizada no compilador PDP-11 BLISS [18], determina o tempo de vida dos temporários e os divide em quatro grupos, observando os seguintes critérios:

1. se eles devem ser alocados a um registrador específico;
2. se eles devem ser alocados a algum registrador de propósito geral;
3. se eles podem ser alocados a um registrador ou à memória;

4. se eles devem residir na memória.

A seguir, ele ordena os temporários alocáveis de acordo com o conjunto de registradores específicos, ou para qualquer registrador, e então tenta uma série de permutações no agrupamento de temporários para alocá-los aos registradores ou à memória, de acordo com o grupo, dando preferência aos registradores.

As abordagens locais de alocação de registradores não atuam no programa como um todo, atuando em pequenas partes do código, o que não gera um resultado muito eficiente.

A partir da década de 80, vários métodos globais de alocação de registradores foram desenvolvidos [7, 8, 6, 15] e são os utilizados atualmente, sendo o mais usado o de Coloração de Grafos [7]. Esta abordagem representa os conflitos entre os valores vivos [1] com um dado número de cores, que representa o número de registradores disponível na máquina. Os valores que não possuem cores são derramados para a memória. Este trabalho foi posteriormente estendido por várias outras contribuições, entre elas [4] e [14]. Apesar de ser o método mais utilizado atualmente para a alocação de registradores, a coloração de grafos apresenta alguns problemas. Para se calcular o custo de se derramar um valor para a memória, é considerado um único bloco básico do programa, não considerando a análise do grafo de fluxo de controle do programa, nem análise do fluxo de seus dados exceto em relação às variáveis vivas. Este é o motivo pelo qual o método não produz código eficiente em algumas situações. Logo, outras técnicas de alocação de registradores mais eficientes quanto a algum critério têm sido pesquisadas e desenvolvidas [4, 14, 6, 8, 23]. As alocações globais de registradores são abordagens intraprocedurais, ou seja, operam dentro de um procedimento do código como um todo. Estes métodos são discutidos detalhadamente na Seção 3. A abordagem proposta nesta dissertação é uma abordagem global e é um método intraprocedural.

Existem também métodos interprocedurais de alocação de registradores, ou seja, a alocação é feita entre os procedimentos do código. A maioria das abordagens interprocedurais são feitas em tempo de compilação. Existe uma abordagem para se fazer alocação interprocedural de registradores em tempo de montagem, desenvolvida por Wall [24], que foi motivada por duas observações importantes. A primeira delas é que se o código gerado está completo, no sentido de que não precisa de alocação de registradores para executar corretamente, então a informação para alocação de registradores feita em tempo de montagem pode ser codificada como informação para realocação de código objeto, e pode ser aplicada ou não. A segunda observação é que se dois caminhos em um grafo de chamada de procedimentos de um programa *(call graph)* não têm interseção, então os mesmos registradores podem ser livremente alocados em cada caminho.

A abordagem para gerar as informações necessárias para a alocação de registradores em tempo de montagem é produzir triplas como código intermediário, e então, em um passo para trás *(backward)* no grafo de fluxo de controle, em direção a cada bloco básico marcar:

- os operandos v que podem ser usados novamente depois que suas variáveis associadas são armazenadas;
- os operandos que foram armazenados e que podem ser usados novamente;

[1] valores que ainda serão utilizados pelo programa

- os operandos restantes.

A seguir, o módulo objeto é gerado, e uma lista de procedimentos é armazenada dentro dele, e, para cada procedimento, uma lista de suas variáveis locais, as variáveis referenciadas por ele, e os procedimentos chamados por ele, juntamente com uma estimativa do número de vezes que cada variável é referenciada e o número de chamadas para cada procedimento. O alocador de registradores então é chamado, e constrói um grafo acíclico dirigido (*(DAG)*) de chamada de procedimentos com um nodo para cada procedimento e coleta a informação do uso dos procedimentos e variáveis. Depois de se estimar o número total de chamadas de cada rotina, o alocador percorre o grafo em profundidade, formando grupos de variáveis que serão atribuídas a registradores. Então, o alocador ordena os grupos por frequência, e assumindo que existem R registradores disponíveis para alocação, ele assinala os R maiores grupos a registradores, e reescreve o código objeto, considerando a informação da alocação.

Três exemplos de abordagens de alocação de registradores interprocedurais são discutidas a seguir. A primeira, desenvolvida por Santhanam e Odnert [21], estende a alocação de registradores por coloração de grafos fazendo computação de variáveis vivas interproceduralmente para determinar os candidatos à alocação para variáveis globais. Ele particiona o grafo de chamada de procedimentos em *webs* (veja Seção 3.1), uma para cada variável local e aloca a registradores aquelas com alta frequência de uso, deixando os registradores restantes à alocação intraprocedural.

Outro método, desenvolvido por Chow [9], é uma extensão da alocação de registradores por coloração de grafos baseada em prioridade [8]. Seu objetivo principal é minimizar o *overhead* causado pelo armazenamento e carregamento de registradores em chamadas de procedimentos. Para isso, ele utiliza a otimização denominada encolhimento *(Shrink Wrapping)* e uma busca em ordem final (*postorder*) no grafo de chamada de procedimentos considerando o uso de registradores nas chamadas de procedimentos. O método supõe que todos os registradores são salvos pelo módulo chamador (*caller-saved*) e adia o armazenamento e carregamento de registradores na chamada tanto quanto possível.

A última abordagem interprocedural, desenvolvida por Steenkiste e Hennessy [22], aloca os registradores em um método similar ao de Wall [24], porém feito em tempo de compilação. Foi desenvolvido para linguagens nas quais os programas consistem em vários pequenos procedimentos, como por exemplo LISP, fazendo com que alocação de registradores interprocedural seja bastante importante. O método efetua uma busca em profundidade no grafo de chamada de procedimentos, e aloca registradores de forma *bottom up* utilizando o princípio de que procedimentos em diferentes subárvores do grafo de chamada podem compartilhar os mesmos registradores. Ele utiliza em cada local de chamada de procedimento as informações disponíveis sobre o uso dos registradores.

Na prática, as abordagens de alocação interprocedurais são complexas e pouco utilizadas.

Nesta seção foram descritas algumas abordagens existentes para a alocação de registradores, uma fase crítica e muito importante na otimização de compiladores. Novos métodos para alocação estão em constante estudo e desenvolvimento. Métodos eficientes foram descobertos, porém todos continuam falhos quanto a algum critério: as abordagens locais não consideram a estrutura do programa como um todo, as abordagens interprocedurais são complexas e pouco usadas, e as

abordagens intraprocedurais ou globais não têm o defeito das abordagens locais, porém são mais complexas e custosas que estas, e não geram código eficiente em algumas situações, como discutido nesta seção. Cada abordagem global [7, 8, 6, 4, 14] é eficiente quanto a algum critério: tempo de execução, número de variáveis computadas, instruções de acesso à memória geradas, mas ainda possuem falhas, como será discutido em mais detalhes na Seção 3.

Este trabalho de dissertação se concentra no problema de Alocação Global de Registradores. Um novo método intraprocedural foi estudado e implementado. Nele são apresentados os resultados de implementação e análise de uma nova heurística para alocação de registradores realizada a partir do algoritmo proposto por Ottoni e Araújo [12], o qual apresenta uma formulação para resolver o problema de alocação global de registradores usando uma técnica denominada Crescimento de Domínios Ativos (*Live Range Growth*). Esta técnica tenta contornar os problemas da Coloração de Grafos [7] fazendo: (a) análise de fluxo de controle de programa, (b) análise de variáveis vivas, e (c) análise de fluxo de dados denominada Análise de Alcançabilidade e Consistência de Registradores (*Register Consistency Reachability, RCR*). Para a implementação, tomou-se como base uma arquitetura load/store, tipo RISC, ou seja, todas as computações são feitas através de registradores, sendo as únicas instruções de acesso a memória justamente as de *load* e *store*.

O texto desta dissertação está organizado da seguinte forma: o Capítulo 2 apresenta o estado da arte em relação aos métodos globais (intraprocedurais) de Alocação de Registradores. O Capítulo 3 descreve o método de Alocação Baseado em Crescimento de Domínios Ativos como proposto por Araújo e Ottoni [12]. O Capítulo 4 apresenta a implementação do algoritmo de Alocação de Registradores Baseada em Crescimento de Domínios Ativos, descrevendo o que foi feito neste trabalho, com a inclusão da Combinação de Registradores. O Capítulo 5 faz uma avaliação dos resultados da implementação do método, assim como a sua comparação com o método de George e Appel [14]. Por fim, os Capítulos 6, 7 e 8 dizem respeito a, respectivamente, conclusão, trabalhos futuros e referências bibliográficas. O Apêndice A apresenta códigos de programas teste para ilustrar a entrada do método de alocação implementado, assim como a saída gerada.

Capítulo 2

Alocação Global de Registradores

2.1 Introdução

Nesta seção são apresentadas diversas técnicas de alocação global de registradores. As técnicas são intraprocedurais e são as principais técnicas desenvolvidas desde os anos 70. Grande parte delas têm como base a heurística de coloração de grafos desenvolvida por Chaitin [7], apresentando também seus problemas.

2.2 Método de Chaitin

O fato de a alocação global de registradores ser vista como um problema de coloração de grafo foi reconhecido por John Cocke em 1971, porém nenhum alocador foi desenvolvido e implementado com esta abordagem até 1981, quando G. Chaitin o fez para o compilador experimental IBM 370 PL/I [7].

A partir daí, este é o método mais adotado atualmente. Esta abordagem oferece uma abstração simplificada do problema de alocação de registradores. Nela, é construído um grafo denominado grafo de interferência, onde seus nodos representam os candidatos à alocação, que podem ser variáveis, temporários ou grandes constantes e os registradores de máquina, e suas arestas representam interferências ou conflitos entre os nodos. Se dois candidatos estão vivos no mesmo ponto de uma rotina, é dito que eles interferem e não podem ocupar o mesmo registrador. Seja k o número de registradores disponíveis na máquina, se os nodos no grafo puderem ser coloridos com k ou menos cores, de tal modo que um par de nodos conectados por uma aresta recebam cores diferentes, então a coloração corresponde à alocação. Se uma coloração com k cores não pode ser efetuada, o código deve ser modificado para se tentar uma nova coloração. Este método é dividido em sete fases:

1. Determinação de quais objetos devem ser candidatos à alocação, ou seja, determinação das *webs* ;
2. Construção do Grafo de Interferência;

3. Transformação de Combinação de Registradores *(Register Coalesce)*;
4. Cálculo do custo de Derramamento *(spill costs)*;
5. Simplificação *(Simplify)*;
6. Seleção *(Select)*;
7. Inserção do código de Derramamento.

O primeiro passo do método é a determinação de quais objetos devem ser candidatos à alocação de registradores. No lugar de se usar simplesmente as variáveis aptas a um registrador, os candidatos são objetos denominados *webs*. Uma *web* é a máxima união de cadeias du[1], tais que para cada definição d e uso u, u pertence a cadeia du de d ou existe $d = d_0$, u_0, ..., d_n, $u_n = u$, tal que, para cada i, u_i está na cadeia du de d_i e de d_{i+1}. Primeiro as cadeias du são construídas computando as definições alcançáveis [18], e então as cadeias du que se interceptam, contendo um uso em comum, são combinadas. A utilização de *webs* no lugar de variáveis para os candidatos à alocação é vantajosa pois um mesmo nome de variável pode ser usado repetidamente em uma rotina para propósitos não relacionados, como por exemplo, o uso de i como índice de laço. Este mesmo i pode ser utilizado como índice em vários laços na mesma rotina. Se o mesmo registrador fosse alocado a todas as referências a i, a alocação teria várias restrições. O uso de *webs* também facilita o mapeamento de variáveis a registradores simbólicos: cada *web* é equivalente a um registrador simbólico.

Após a computação das *webs*, o próximo passo é a construção do Grafo de Interferência. Nele, existe um nodo para cada registrador de máquina e um nodo para cada *web*. Os registradores de máquina devem ser nodos porque os registradores não são homogêneos, pois existem as convenções de chamadas a procedimentos *(calling conventions)* e de estrutura de pilha que necessitam que alguns registradores sejam reservados a elas. Uma aresta entre dois nodos no grafo de interferência existe se um deles estiver vivo no ponto de definição do outro nodo. Se existisse uma aresta entre dois nodos por eles estarem vivos simultaneamente, o número de cores necessárias para se colorir o grafo seria alto, existiria mais arestas do que o necessário no grafo [18]. Por isso, basta que exista uma aresta entre dois nodos se um deles estiver vivo no ponto de definição do outro. O número de arestas que conecta um nodo a outros é denominado nível do nodo.

Após a construção do grafo, uma transformação conhecida como Combinação de Registradores *(Register Coalesce)* é aplicada no código intermediário. Esta transformação é uma variação da propagação de cópia que elimina cópias de um registrador para outro. Ela procura no código intermediário por instruções de cópia entre registradores, como $s_j \leftarrow s_i$, tais que, s_i e s_j não interferem entre si e não são armazenados na memória entre a atribuição da cópia e o final da rotina. Depois de se achar tais instruções, a transformação de combinação de registradores procura as intruções que escrevem em s_i e as modifica para que escrevam em s_j, remove a instrução de cópia, e modifica o grafo de interferência combinando s_i e s_j em um único nodo que interfere com todos os nodos que s_i e s_j interferiam individualmente. Esta transformação é bastante poderosa pois:

[1] *du* signfica *d*efinição e *u*so

- simplifica vários passos no processo de compilação, tal como a remoção de cópias desnecessárias introduzidas no caso do programa ter sido traduzido para a forma SSA [11];
- pode ser utilizada para garantir que os valores dos argumentos são transportados para os registradores próprios antes da chamada do procedimento;
- possibilita que instruções de máquina com registradores alvo e fonte especificados tenham seus operandos e resultados em seus devidos lugares;
- possibilita que instruções de dois endereços, como as encontradas em arquiteturas CISCs, tenham seus registradores alvos e operandos manipulados apropriadamente;
- permite que instruções que necessitam de um par de registradores para algum operando ou resultado, sejam a eles atribuídas.

O próximo passo da alocação de registradores por coloração de grafos é a computação dos custos de derramamento e de carregamento do conteúdo dos registradores, caso não seja possível a alocação de todos os candidatos diretamente a registradores. Se as decisões de derramamento forem feitas com cuidado, elas podem fazer com que o grafo seja colorido com R cores e que poucas instruções de *load* e *store* sejam inseridas no código. O derramamento do conteúdo de um registrador para a memória e seu posterior carregamento da memória, quando for necessário, faz o grafo ser colorido com R cores, e tem o efeito de potencialmente dividir uma *web* em duas ou mais *webs*, reduzindo as interferências do grafo, o que aumenta a chance do resultado ser colorido com R cores.

O custo do derramamento de uma *web* w é calculado como [18]:

$$defwt * \sum_{def \in w} 10^{depth(def)} + usewt * \sum_{use \in w} 10^{depth(use)} - copywt * \sum_{copy \in w} 10^{depth(copy)}$$

onde *def*, *use* e *copy* são definições, usos e cópias de registradores na *web* w ; *defwt*, *usewt* e *copywt* são pesos relativos associados aos tipos de instruções; e *depth* é a profundidade do aninhamento de laço. O custo de derramamento deve considerar os seguintes fatores:

1. se o valor de uma *web* pode ser mais eficientemente recomputado do que carregado, o custo da recomputação é usado;
2. se o destino ou fonte de uma instrução de cópia é derramado, a instrução não é mais necessária;
3. se o valor derramado é usado várias vezes no mesmo bloco básico e o valor restaurado continua vivo até seu último uso no bloco, uma única instrução de *load* é necessária no bloco.

Depois deste passo, tenta-se colorir o grafo de interferência com R cores, onde R é o número de registradores disponíveis. Observa-se que este é um problema NP-completo, além de que o grafo pode não ser colorido com R cores. Devido a estes fatos, é usada uma abordagem para simplificar o grafo.

A abordagem é denominada regra do *grau* $< R$: dado um grafo que contém um nodo de nível menor que R, o grafo é colorido com R cores se e somente se o grafo sem aquele nodo puder ser colorido com R cores. O nível do nodo é o número de arestas do nodo. O fato de a coloração com R cores do grafo inteiro implicar na coloração com R cores do grafo sem o nodo de nível menor que R é óbvio. Por outro lado, suponha que exista uma coloração com R cores do grafo sem o nodo. Já que o nodo tem nível menor que R, existe pelo menos uma cor que não é usada em um nodo adjacente a ele, e que pode ser atribuída a ele. Esta regra, porém não faz com que um grafo arbitrário seja colorido com R cores.

O processo de coloração então começa com a busca repetida no grafo por nodos que possuam menos que R vizinhos. Esta fase do método de alocação é denominada Simplificação. Cada nodo encontrado é removido do grafo e colocado em uma pilha, desta maneira eles serão coloridos de forma inversa a que foram removidos do grafo, ou seja, os nodos com um número maior de vizinhos serão coloridos antes dos nodos com um número menor de vizinhos, sendo que quanto menor o número de vizinhos de um nodo, mais facilmente é encontrada uma cor para ele. Os nodos adjacentes aos nodos removidos são marcados para serem usados na atribuição dos registradores. Se todos os nodos do grafo forem removidos, o grafo pode ser colorido com R cores. Os nodos são então retirados da pilha e a cada um é atribuída uma cor dentre as que não foram atribuídas a nodos adjacentes. Quando a regra do *nível* $< R$ não for aplicável, um nodo com nível $\geq R$ é escolhido para ser derramado. Ele é removido do grafo e marcado para derramamento. Quando o grafo não possuir mais nodos, os nodos que foram marcados para derramamento são armazenados na ou recarregados da memória, inserindo-se código de derramamento no programa, e o processo de alocação é então repetido, até que nenhum código de derramamento seja necessário, e o alocador passa para o passo seguinte, de se atribuir as cores. Este passo é chamado de Seleção.

A seguir, as cores são atribuídas aos nodos, sendo que nodos adjacentes não recebem cores iguais, e cada cor corresponde a um registrador de máquina.

O número de cores necessário para colorir um grafo de interferência é chamado de pressão por registradores (*register pressure*), e as modificações feitas no código para tornar o grafo colorido com R cores são descritas como reduzindo a pressão por registradores. Em geral, o efeito do derramamento de um valor é a divisão da *web* em duas ou mais *webs* e a distribuição das interferências da *web* original entre as novas. Se as tentativas de se colorir o grafo de interferência com R cores falharem, os nodos marcados para derramamento, aqueles que possuem o menor custo de derramamento, são derramados.

Bernstein [3] discute em seu trabalho três heurísticas que podem ser utilizadas para selecionar valores para o derramamento e implementa um alocador que tenta as três e utiliza a que melhor se aplica à situação. A primeira delas é baseada na observação de que o derramamento de uma *web* com nível maior reduz o nível de muitos outros nodos, logo é mais provável de se maximizar o número de *webs* que tenham nível $< R$. As duas outras heurísticas levam em consideração o efeito global de cada *web* na pressão por registradores. Dos 15 programas testados pelos autores [3] com as três heurísticas, a primeira delas foi a mais eficiente em quatro programas, a segunda em 6 e a terceira em 8, e em todos os casos o resultado foi mais eficiente que outras abordagens de alocação que utilizam os custos de derramamento como mostrado no método de Chaitin.

A abordagem de Chaitin não produz código eficiente em certas situações. Isto se deve ao fato de que o método não faz análise do fluxo de controle do programa e nem análise de fluxo de seus

dados. Além disso, para se calcular os custos de derramamento de um valor para a memória, o método considera um único bloco básico, o que não captura o fato de uma variável ser pouco referenciada em um bloco básico, porém muito referenciada no bloco básico sucessor.

Como a análise de fluxo de controle do programa não é feita, se uma variável estiver em um laço e seu custo de derramamento for baixo, ela será derramada e o código ficará ineficiente, pois o código de derramamento deverá ser executado em toda iteração do laço. Isto não aconteceria se a análise do fluxo de controle do programa fosse feita pelo método.

Como a alocação de registradores baseada em coloração de grafos é resolvida por heurística, o processo de coloração é exponencial.

2.3 Método de Chow e Hennessy

Alocação de registradores por coloração de grafo baseada em prioridade é similar à abordagem de Chaitin, mas difere em vários detalhes importantes. Esta abordagem foi proposta por Chow e Hennessy [8], e pretende ser mais sensitiva aos custos e benefícios de decisões individuais de alocação do que a abordagem de Chaitin.

Uma noção de prioridades é adotada nesta abordagem. A atribuição de prioridades é baseada em estimativas dos benefícios derivados da alocação de valores individuais a registradores. Ao se utilizar prioridades, o processo de coloração, que é exponencial, pode ser executado em tempo linear. O algoritmo pode ser parametrizado para suprir características e configurações de registradores de diferentes máquinas.

Uma diferença significativa entre as duas abordagens está no fato de que a abordagem de Chow e Hennessy aloca todos os objetos a endereços de memória antes da alocação de registradores e tenta migrá-los para os registradores, enquanto no método de Chaitin, eles são alocados a registradores simbólicos. Isto faz a abordagem de Chaitin ser otimista, pois supõe que todos os registradores simbólicos poderão ser alocados a registradores reais. Já a alocação baseada em prioridade é pessimista, pois supõe que pode não ser possível alocar todos os objetos a registradores. Devido a esta abordagem pessimista, para uma arquitetura tipo RISC, que não possui operações diretamente na memória, o método reserva quatro registradores de cada tipo para serem utilizados na avaliação de expressões que envolvem variáveis que não puderam ser alocadas a registradores. Portanto, este método começa com um defeito: menos registradores estão disponíveis para alocação. O fato dos objetos serem alocados a endereços de memória antes da alocação de registradores evita a inserção de código de derramamento.

Na alocação de registradores de Chow e Hennessy, o código do programa é dividido em segmentos de código não maiores que um bloco básico, os quais representam a menor extensão do código na qual as variáveis são alocados a registradores. O tempo de execução economizado pela alocação de uma variável a um registrador se refere a quão mais rápido o segmento de código executa quando a variável é atribuída a um registrador.

Três parâmetros são definidos: *movcost*: custo de *move* de memória para registrador ou registrador para memória na máquina alvo; *lodsave*: quantidade economizada de tempo de execução de cada referência a uma variável atribuída a um registrador, em relação a mesma referência residindo na memória; *strsave*: quantidade economizada de tempo de execução de cada definição

de uma variável atribuída a um registrador, em relação a mesma variável residindo em memória. Neste método existe a alocação local de registradores e a alocação global de registradores. A alocação local se refere a um trecho do código do programa, e precede a alocação global. Para cada variável considerada no segmento de código, a economia local ganha com a atribuição de uma variável a um registrador pode ser estimada por:

$$netsave = lodsave * u + strsave * d - movcost * n$$

onde u é o número de usos da variável, d o número de definições e n é 0, 1 ou 2, dependendo do número de instruções de acesso a memória necessárias no início do segmento de código considerado. A quantidade *netsave* de um segmento de código pode ser incerta, pois o código antecessor e sucessor a ele não é considerado. Desta forma, existem duas quantidades, *maxsave* e *minsave*, que representam a economia máxima e mínima, respectivamente, e são determinadas por:

$$maxsave = lodsave * u + strsave * d$$
$$minsave = lodsaave * u + strsave * d - movcost * n$$

A economia total depois da alocação de registradores é um valor entre *minsave* e *maxsave*. O critério para determinar a alocação local de uma variável a um registrador, quando os blocos ao redor do bloco em questão são considerados, é dado por:

$$minsave > movcost * (p+s)$$

onde p é o número de blocos predecessores e s é o número de sucessores do bloco. Quando esta condição é satisfeita, a variável pode ser alocada localmente a um registrador independentemente do resto do programa. A fase de alocação local determina, então, quais variáveis são alocadas a registradores. Porém, a determinação de quais registradores são atribuídos às variáveis é feita na fase de alocação global. Nesta fase, o alocador procura oportunidades para atribuir o mesmo registrador a uma mesma variável por todos os blocos básicos, para desta maneira, diminuir o número de instruções de acesso à memória.

Uma outra diferença entre o método de Chow e Hennessy e de Chaitin está no conceito de *web* e grafo de interferência. Chow e Hennessy representa a *web* de uma variável como o conjunto de blocos básicos nos quais ela está viva e o chama de domínio ativo (*live range*). O domínio ativo é conservativo, porém é menos preciso que as *webs* do método de coloração de grafos, inserindo mais interferência entre os nodos do grafo de interferência, pois no método de coloração de grafos, só existe a interferência quando uma variável está viva no ponto de definição da outra, enquanto que no método baseado em prioridade, existe a interferência quando as variáveis estão vivas no mesmo bloco, não importando o ponto de definição das mesmas.

O processo de alocação é feito de acordo com os custos e economias calculados na fase de alocação local. Cada iteração do passo de alocação global atribui uma variável a um registrador, escolhendo o domínio ativo mais promissor para a alocação de acordo com os custos e economias calculados para cada domínio ativo. Primeiro, os domínios ativos com alta prioridade são atribuídos a registradores, desta maneira, espera-se que o resultado da alocação seja próxima ao

ótimo. O algoritmo termina quando todos os domínios ativos ou partes deles forem alocados ou todos os registradores forem utilizados nos segmentos de código.

O método de Chaitin [7] derrama valores para a memória quando não há registradores suficientes para a alocação. Chow e Hennessy fazem a divisão de domínios ativos quando isto acontece. A divisão de domínios ativos é repetida até que os domínios ativos divididos possam ser coloridos ou até que eles sejam constituídos de segmentos de códigos simples. Se a um domínio ativo dividido não for atribuída nenhuma cor no término da coloração, ele não é alocado a nenhum registrador.

A alocação de registradores no método de Chow e Hennessy é feita em um contexto independente de máquina, utilizando alguns parâmetros que dependem da máquina alvo. Entre os parâmetros utilizados estão os benefícios do acesso a registradores em relação ao acesso à memória. A eficiência do algoritmo não é afetada pelo número de registradores disponíveis na máquina.

O método de Chow e Hennessy gera código eficiente e considera a estrutura de laços do programa. Este fator é importante, pois o método de Chaitin não considera o grafo de fluxo de controle do programa.

Briggs [4] investigou o tempo de execução de dois alocadores de registradores, um desenvolvido por ele baseado em coloração de grafos, e outro baseado em prioridade. Briggs descobriu, em seus testes, que o alocador desenvolvido por ele executava em O($n\ log\ n$), enquanto o baseado em prioridade executava em O(n^2), onde n é o número de instruções do programa.

Um defeito da abordagem de Chow e Hennessy é o fato de reservar quatro registradores de cada tipo para serem usados na avaliação de expressões que envolvem variáveis que não podem ser alocadas a registradores. Em uma arquitetura que possui registradores de propósito geral e de ponto flutuante, esta reserva significa oito registradores a menos disponíveis para a alocação. Registradores já são um recurso escasso, oito registradores é um número significante de registradores a menos para serem utilizados pela alocação.

Outro defeito é o tempo de execução do método ser maior que o de coloração de grafos, como mostrado na pesquisa de Briggs.

2.4 Método de Briggs

Preston Briggs [4] desenvolveu um trabalho que acrescenta várias melhorias e extensões ao trabalho de Chaitin. Ele pode ser resumido basicamente em quatro resultados:

1. Coloração Otimista (*Optimistic coloring*): o problema de alocação de registradores baseado em coloração de grafos é um problema NP-completo. Devido a este fato, os pesquisadores concentram seus esforços na determinação de abordagens baseadas em heurísticas para a alocação de registradores, existindo, portanto, pouco estudo na área de uma coloração otimista para o problema. A abordagem de Chaitin assume pessimisticamente que qualquer nodo de nível maior que o número de cores disponíveis não é colorido e deve ser derramado. Briggs assume otimisticamente que todos os nodos com nível maior também recebem cores, atingindo com isso custos de derramamento menores e códigos mais rápidos. Ele modificou a fase de Simplificação do algoritmo de Chaitin: nesta fase, os nodos com nível $< R$ são removidos do grafo, até que todos os nodos restantes tenham nível $\geq R$. Quando isto

acontece, um candidato ao derramamento é escolhido e removido do grafo, porém ele não é marcado para o derramamento, mas sim colocado na pilha na esperança de que alguma cor fique disponível para ele, apesar de seu nível maior. Desta maneira, os candidatos a derramamento são também incluídos na pilha de coloração. Na próxima fase do processo de alocação, se nenhuma cor estiver disponível, ele não é colorido. Se todos os nodos receberem cores, a alocação é bem sucedida, se existirem nodos não coloridos, é inserido código de derramamento para tais nodos, e o processo de alocação é repetido;

2. Coloração de Pares (*Coloring pairs*): muitas arquiteturas de máquina utilizam pares de registradores de precisão simples para armazenar valores de precisão dupla. O pessimismo da abordagem de Chaitin é enfatizado ao se fazer a tentativa de colorir pares de registradores. O grafo de interferência e o modo como o alocador o interpreta devem ser modificados com a introdução de pares de registradores, distorcendo o alocador, que derrama valores em vários casos onde existem registradores disponíveis para eles. Briggs propõe um método mais satisfatório para se lidar com pares de registradores, pois ele adia as decisões de derramamento para a fase de atribuição dos registradores, somente derramando um valor quando se descobre que não existe uma cor para ele;

3. Rematerialização (*Rematerialization*): rematerialização é o processo de regenerar em valores de registradores constantes que são mais eficientemente recomputadas do que derramadas e recarregadas. Briggs desenvolveu um método para melhorar a rematerialização. Este método envolve dividir uma *web* nos valores que a formam, fazer uma computação de fluxo de dados que associa a cada valor potencialmente rematerializável a instrução que seria usada para rematerializá-lo e então construir novas *webs*, cada uma consistindo dos valores que têm a mesma instrução associada a ele. A informação de rematerialização é propagada pelo grafo SSA [11], com isso, uma classe maior dos valores rematerializáveis é descoberta e isolada;

4. Divisão de *Web* (*Web Splitting*): é uma abordagem que leva em consideração a estrutura do grafo de fluxo de controle do procedimento para agressivamente dividir *webs* antes da coloração. A divisão de *webs* em vários pedaços e a consideração destes pedaços em separado produz um grafo de interferência que pode ser colorido com menos cores.

Briggs também desenvolveu uma heurística de combinação de registradores conservativa, em contraste a heurística de *coalesce* agressiva de Chaitin, onde um grafo colorido com R cores não pode ser colorido com R cores após a aplicação da transformação. A heurística conservativa de Briggs resolve este problema, porém deixa muitas instruções de cópia desnecessárias no código.

O método de alocação desenvolvido por Briggs continua calculando os custo de derramamento considerando um único bloco básico e não faz análise de fluxo de controle. Logo, é um método que melhora certos aspectos do método de Chaitin, mas ainda conserva vários de seus defeitos.

2.5 Método de Callahan e Koblenz

Callahan e Koblenz [6] desenvolveram uma abordagem hierárquica para alocação global de registradores via coloração de grafos. Esta abordagem decompõe o grafo de fluxo de controle em uma árvore de *tiles*, colore cada *tile* individualmente, e combina os resultados em dois passos ao longo da árvore. Representa uma tentativa de se ganhar a precisão da abordagem de Chaitin para alocação juntamente com uma abordagem estrutural para divisão do domínio ativo.

No método de Chaitin [7], a decisão de se derramar uma variável não é baseada na estrutura de controle de fluxo do programa, e uma vez que uma variável é derramada para a memória, todas as referências a esta variável devem buscar seu valor da memória e todas as suas definições devem guardar o novo valor na memória. Isto gera código menos eficiente quando variáveis são derramadas em porções de código muito executadas, como laços. Na abordagem de Callahan e Koblenz, o derramamento ocorre em porções pouco executadas do código. O método também permite que uma variável seja atribuída a um registrador em uma parte do programa, à memória em outra parte do programa, e a um registrador diferente em uma terceira parte do programa.

A estrutura hierárquica de um programa é representada como uma árvore, porque nesta representação é mais fácil separar áreas de maior e menor frequência de execução. Código de derramamento é inserido nas partes menos frequentemente executadas do código do programa.

A árvore *tile* representa a estrutura do programa, e algumas heurísticas são utilizadas para construir a árvore a partir do grafo de fluxo de controle do programa [6]. Cada par de conjuntos em um *tile* T são disjuntos ou então um é um subconjunto próprio do outro.

Uma variável v é local a um *tile* t se todas as referências a v são feitas em blocos dentro de t e v não está viva em nenhum sucessor ou predecessor de t. Referências a uma variável que está presente em t, porém não é local a ele é chamada de global em relação àquele *tile*.

O processo de alocação baseado no método de Callahan e Koblenz tem duas fases. Na primeira fase, *tiles* são visitados em uma ordem *bottom-up*, um grafo de interferência local é construído e colorido para cada *tile*. Desta forma, registradores físicos e pseudo-registradores são alocados de uma forma *bottom-up* na árvore, usando o método de coloração de grafos. As decisões de derramamento são feitas e um sumário da alocação é passado para o pai do *tile* e incorporado a seu grafo de interferência.

Se $t_2 \subset t_1$ e não existe $t \in T$ tal que $t_2 \subset t \subset t_1$, então diz-se que t_2 é filho ou *subtile* de t_1 e t_1 é o pai de t_2, denotado por $t_1 = parent(t_2)$. O conjunto *blocos(t)* é o conjunto de blocos básicos que são membros de t, porém não são membros de nenhum filho de t. Após todo o *tile* ter sido alocado a pseudo-registradores, a segunda fase da alocação percorre a árvore de cima para baixo *(top-down)* mapeando pseudo-registradores a registradores físicos e atualizando as decisões de derramamento, inserindo código de derramamento quando desejado ou necessário, porém não necessariamente onde a decisão de derramamento foi feita. Observa-se que quando determinados valores devem estar em um registrador físico apropriado, essas variáveis são atribuídas a estes registradores durante a primeira fase.

Duas decisões são importantes para o derramamento:

- quais variáveis devem ser derramadas;
- onde o código de derramamento deve ser inserido.

O melhor lugar para a inserção de código de derramamento pode não ser onde a decisão de derramamento foi tomada, como acontece no método de Chaitin [7]. No método de Callahan e Koblenz, a decisão de derramamento é tomada na primeira fase, porém a inserção de código de derramamento é feita na segunda fase do alocador.

Para se determinar quais variáveis devem ser alocadas a registradores e onde inserir código de derramamento, as seguintes fórmulas são utilizadas:

$$Local\_weight_t(v) = \sum_b Prob(b) * Ref_b(v)$$
$$Transfer_t(v) = \sum_e Prob(e) * Live_e(v)$$
$$Weight_t(v) = \sum_s (Reg_s(v) - Mem_s(v)) + Local\_weight_t(v)$$
$$Reg_t(v) = Reg?_t(v) * min(Transfer_t(v), Weight_t(v))$$
$$Mem_t(v) = Mem?_t(v) * Transfer_t(v)$$

onde e percorre todas as arestas da árvore, b representa todos os blocos em *blocks(t)*, e s passa por todos os *subtiles* do *tile t*. $Live_e(v)$ é 1 se a variável v está viva ao longo da aresta e e 0 caso contrário. *Prob(b)* é a probabilidade de b ser executado e *Prob(e)* é a probabilidade de existir um fluxo ao longo da aresta e. $Ref_b(v)$ é o número de referências a variável v no bloco b. $Reg?_t(v)$ é 1 se a variável v foi alocada a um registrador no *tile* t e 0 caso contrário. $Mem_t(v)$ é 1 se a variável v não foi alocada a um registrador no *tile* t, e 0 caso contrário. $Local\_weight_t(v)$ corresponde a economia de se deixar v alocado a um registrador nos blocos existentes em *tile t*. $Transfer_t(v)$ é o custo de derramamento e/ou recarga da variável v na entrada e na saída do *tile t*. $Weight_t(v)$ é usado para guiar a heurística de qual variável deve ser derramada, sendo baseado no número de usos da variável no *tile*, na penalidade de se alocar v à memória neste *tile*, se ele é alocado a um registrador em algum de seus *subtiles*, e, na penalidade de se alocar v a um registrador neste *tile* se ele deve ser derramado em um dos seus *subtiles*. $Reg_t(v)$ é a penalidade de se ter o *parent(tile)* alocando v à memória quando o *tile* o aloca a um registrador. $Mem_t(v)$ é a penalidade de se ter o *parent(tile)* alocando v a um registrador quando o *tile* o alocou à memória.

Essas equações são usadas na primeira fase do alocador para determinar quais variáveis devem ser alocadas a um registrador e para se determinar se é interessante alocar uma variável a um registrador mesmo se seu *parent(tile)* pode alocar a variável a um registrador. Elas também são utilizadas na segunda fase do alocador para se determinar quando a alocação de uma variável a um registrador deve ser trocada para uma alocação à memória quando a variável reside na memória no *parent(tile)*. Quando $weight_t(v) < 0$, v não deve ser alocada a um registrador. Quando $transfer_t(v) + weight_t(v) < 0$ v é marcada para não ser alocada a um registrador.

Na segunda fase do alocador, o código de derramamento deve ser inserido em locais apropriados:

- quando v está na memória em um *tile* t e em um registrador no *parent(t)*, código para mover v para a e da memória nas arestas que entram e saem de t, respectivamente;
- quando v está no registrador em t e na memória em *parent(t)*, e se $weight_t(v) > transfer_t(v)$, transferências da memória para o registrador são geradas, caso contrário, a alocação de v em t é substituída para refletir o fato de que ele deve estar na memória.

As diferenças do método de Callahan e Koblenz [6] para o de Chaitin [7] são então, que o primeiro se benefica dos padrões de uso local de uma variável e insere código de derramamento inteligentemente. O método de Callahan e Koblenz é um esquema eficiente, que pode ser executado em paralelo, diminuindo o tempo de execução do processo de alocação.

Apesar disso, como a alocação no método de Callahan e Koblenz é feita utilizando a coloração de grafos, os mesmos problemas existentes no método de Chaitin existem neste método, como por exemplo, a inexistência de análise de fluxo de controle do programa.

2.6 Método de Traub, Holloway e Smith

Traub, Holloway e Smith [23] propuseram um algoritmo, denominado *liner scan*, que direciona a alocação global de candidatos à alocação para registradores baseado em uma varredura linear simples ao longo do programa sendo compilado. Esta abordagem faz sentido para sistemas nos quais a velocidade de compilação é importante. Experimentos com este algoritmo demonstram que ele é muito mais rápido do que o método de coloração em *benchmarks* com um número grande de candidatos à alocação.

O método de Traub et. al se inicia com o cálculo do intervalo de vida de um candidato à alocação *(lifetime interval)*. O intervalo de vida de um candidato à alocação é o segmento do programa que começa com a primeira referência ao candidato no código e termina com a última referência a ele, ou seja, quando ele não está mais vivo. O alocador visita cada intervalo de vida, de acordo com sua ocorrência no código do programa, e considera quantos intervalos estão ativos. O número de intervalos ativos simultaneamente representa a competição para os registradores de máquina disponíveis neste ponto do programa. Quando existem muitos intervalos de vida ativos ao mesmo tempo, uma heurística simples escolhe quais são derramados para a memória, e a alocação continua. O método tenta detectar e resolver os conflitos localmente, fazendo com que ele seja mais rápido que a coloração de grafo, que tenta resolver os conflitos da unidade sendo compilada como um todo.

Um fato importante do método de Traub et. al é que ele aloca os registradores aos candidatos e reescreve o código, inserindo as instruções de acesso à memória em um único passo no código. Esta propriedade introduz certa flexibilidade no processo de alocação, pois os candidatos que são derramados para a memória têm várias chances de serem alocados a registradores durante seu intervalo de vida. Para tal, um segundo passo pelo código do programa é necessário.

Os candidatos à alocação são chamados de temporários, e podem ser variáveis e temporários gerados pelo compilador. Ao se examinar o intervalo de vida dos temporários, foi observado que ele pode conter um ou mais intervalos nos quais nenhum valor é mantido. Esses intervalos são chamados de buracos de intervalo de vida *(lifetime holes)*. Se r foi atribuído ao temporário t durante todo o intervalo de vida de t, um outro temporário u pode ser atribuído a r durante o intervalo de vida de t se o intervalo de vida de u couber dentro de um buraco de intervalo de vida de t. O método realiza um passo pelo código do programa para computar os intervalos de vida dos temporários e os seus buracos, ordenando-os linearmente de acordo com os blocos básicos do programa.

A fim de minimizar o número de registradores de máquina sendo alocados, o método atribui

dois intervalos de vida que não se cruzam a um mesmo registrador, e também atribui dois temporários a um mesmo registrador quando o intervalo de vida de um está inteiramente contido no buraco do intervalo de vida do outro.

O alocador visita o código, processando os temporários de acordo com a ordem em que eles são encontrados no código. O processamento de um temporário t envolve a alocação de t a um registrador, caso t ainda não tenha sido atribuído a nenhum registrador. Um registrador disponível para alocação pode ser visto como contendo um buraco de intervalo de vida que se estende até um ponto do programa onde ele não está mais disponível. A seleção de um registrador para ser alocado a t consiste na procura por um registrador cujo buraco de intervalo de vida seja grande o bastante para conter todo o intervalo de vida de t. Se existirem muitos registradores com buracos de intervalo de vida grande o bastante para conter o intervalo de t, o registrador cujo buraco de intervalo seja menor é escolhido heuristicamente para ser alocado a t. Ao se alocar t a um registrador, todas as suas referências são substituídas pelo registrador atribuído a ela.

Se em determinado ponto do programa existirem mais intervalos de vida de temporários que registradores de máquina disponíveis para alocação, alguns destes valores são derramados para a memória. Um alocador tradicional baseado em varredura linear primeiramente visita os intervalos de vida dos temporários ordenados, decidindo quais temporários devem residir em registradores e quais devem residir na memória. A seguir, em uma segunda fase, o alocador passa pelo código do programa e reescreve cada operando com a referência ao registrador alocado a ele ou à memória. Traub, Holloway e Smith observaram que a alocação e a reescrita do código podiam ser feitas em um único passo pelo código do programa. Quando um temporário t é encontrado pela primeira vez no código, o processo de reescrita é parado e a alocação para t é efetuada. Se algum outro temporário tiver que ser derramado para criar um registrador disponível para t, um temporário u que reside em um registrador é derramado para a memória e t é atribuído a este registrador r. Essas decisões de derramamento são baseadas em uma heurística de prioridade que compara a distância de cada temporário a sua próxima referência, pesando também a profundidade do laço no qual o temporário se encontra. O temporário de menor prioridade é escolhido para o derramamento.

Desta maneira, um ponto de derramamento marca uma divisão no intervalo de vida do temporário derramado u. Isto porque todas as referências a u até este ponto já foram reescritas para utilizar o registrador r, e elas não serão mudadas: as decisões de derramamento afetam somente as futuras referências a u. Quando uma posterior referência a u é encontrada, u deve ser alocado a um registrador. Se a referência for um uso de u, um registrador r que esteja disponível é encontrado e uma instrução de *load* u para o registrador r é inserida. Neste caso, u fica atribuído a r até que algum temporário de maior prioridade faça com que u seja derramado, ou então seu intervalo de vida termine, dividindo-se o intervalo de vida de u novamente. O benefício desta abordagem é que u não tem que ser recarregado se uma referência a u for feita posteriormente. As referências futuras a u são otimisticamente planejadas. Se a próxima referência a u for uma definição, u é alocado a um registrador, porém o armazenamento de seu novo valor na memória é adiado até que algum outro temporário force u a ser derramado. Todas as referências a u são reescritas para usarem r, e se o fim do intervalo de vida de u for alcançado, um *store* pode não ser inserido.

Esta abordagem otimista de se lidar com temporários derramados é chamada de segunda

chance, pois é dada uma segunda chance para que o temporário seja alocado a um registrador. Esta abordagem então, disponibiliza um registrador diferente para cada divisão no intervalo de vida de um temporário.

A fim de se otimizar a alocação para que uma instrução de *store* de um temporário seja economizada se o valor do temporário presente no registrador for igual a seu valor armazenado na memória, a informação a respeito da consistência do valor de r com a memória é mantida pelo alocador.

A abordagem da segunda chance traz um custo à alocação: a criação de conflitos nas alocações presumidas na fronteira do bloco básico. Para que a semântica do programa permaneça correta, a fase da alocação e reescrita é seguida por uma visita transversal às arestas do grafo de fluxo de controle CFG do programa, na qual os conflitos são resolvidos inserindo-se um conjunto apropriado de instruções de *load*, *store* e *move*. Um mapa com informações a respeito da localidade de um temporário no início e no fim de cada bloco básico é mantido pelo alocador. Se um temporário está atribuído a um registrador no final do bloco básico predecessor, mas na memória no início do bloco básico sucessor, uma instrução de *store* é inserida. Se um temporário moveu da memória para um registrador, uma instrução de *load* é inserida. Se um temporário está em dois registradores diferentes entre uma aresta do CFG, uma instrução de *move* é inserida. Desta forma, a semântica do programa é garantida. Esta fase é denominada Resolução.

Na fase de Resolução, uma análise de fluxo de dados é feita para garantir a correção do código do programa. Nela, um *bit* é ligado para cada temporário t se t está alocado a um registrador r e está consistente com seu valor na memória. Uma definição de t religa o *bit*, enquanto um derramamento de t liga o *bit*. Se o *bit* estiver ligado durante o derramamento de t, a instrução de *store* não é gerada.

O resultado do método de Traub, Holloway e Smith [23] foi comparado ao método de coloração de grafos. O método de Traub et. al é mais rápido, podendo habilitar otimizações interprocedurais de grandes programas e sendo apropriado para geração de código em tempo de execução.

Um defeito da abordagem de Traub, Holloway e Smith é não fazer análise de fluxo de controle do programa.

Um outro problema é que o custo de derramamento de uma variável é calculado por uma heurística de prioridade, que compara a distância do temporário a sua próxima referência. Esta maneira de se calcular o custo de derramamento não é eficiente, pois uma variável pode ser definida em um ponto, porém pode ser usada somente em um ponto distante de sua definição e usada freqüentemente depois disso.

2.7 Método de Knobe e Zadeck

Knobe e Zadeck [17] desenvolveram e avaliaram um método de alocação de registradores que usa uma árvore de controle de procedimentos para guiar as decisões de derramamento e de simplificação do grafo de interferência. Neste método, as porções do programa nas quais o número de candidatos à alocação é maior do que os registradores disponíveis são simplificadas, armazenando os valores na memória na entrada da região e recarregando-os na saída da região. A simplificação é possível até que o número de candidatos não exceda o número de registradores

disponíveis. Quando este problema (número de candidatos superior ao número de registradores diponíveis) for resolvido e quando um candidato não "couber"em um registrador, ele é dividido em dois. O custo desta divisão é a inserção de instruções de *move* entre as regiões divididas. É importante que se selecione, além do candidato apropriado à divisão, a localidade apropriada à divisão, a fim de minimizar o custo das instruções de *move*. Tanto a simplificação quanto a divisão de candidatos utilizam uma árvore de controle, que considera a estrutura do programa na determinação dos custos. O algoritmo é denominado Alocação de Registradores Baseada em Árvore de Controle *(Control-Tree Register Allocation)*.

Como utiliza a coloração de grafos, possui os mesmos problemas que este método.

2.8 Método de Gupta, Soffa e Steele

Apesar de o método de coloração de grafos ser a abordagem mais utilizada para alocação de registradores atualmente, não só seu tempo de execução pode ser alto em algumas situações como também pode utilizar muita memória se o grafo de interferência se tornar muito grande.

Baseados nos resultados conseguidos por R. Tarjan, que diz respeito à possibilidade de se colorir grafos decompostos usando separadores de *cliques*, Gupta, Soffa e Steele [15] desenvolveram um algoritmo para alocação de registradores via separadores de *cliques*. Primeiramente, este algoritmo separa o código do programa em segmentos de código utilizando a noção de separadores de *cliques*. A seguir, o grafo de interferência para cada segmento é construído e colorido independentemente, um por vez. As colorações de todas as partições são então combinadas para se obter a alocação de registradores para o programa inteiro. Esta técnica é eficiente tanto em tempo quanto em espaço, pois um grafo de interferência de um segmento do código do programa deve ser construído e utilizado em um dado ponto do algoritmo. Além disso, a partição do grafo utilizando o conceito de separadores de *cliques* aumenta a probabilidade de se obter uma coloração sem derramamento de variáveis, fazendo com que a alocação seja mais eficiente. Para um código sem desvios, uma alocação ótima pode ser obtida utilizando-se este método. Já para códigos com desvios (*branches*), uma alocação ótima pode ser obtida para um dado caminho, geralmente o mais freqüentemente executado, e uma alocação quase ótima pode ser obtida para os caminhos alternativos.

Um separador de *clique* é um subgrafo completamente conectado, cuja remoção desconecta o grafo em pelo menos dois subgrafos. Estes subgrafos podem ser posteriormente decompostos em subgrafos menores usando separadores de *cliques*. Um separador de *clique* é máximo se não existe um nodo no grafo que pode ser adicionado ao separador de *clique* para produzir um *clique* maior. Máximos separadores de *clique* com no máximo R nodos têm duas propriedades interessantes: eles dividem o programa em segmentos para os quais a alocação de registradores pode ser efetuada em separado, e podem ser construídos examinando-se o código. O separador de *clique* fica presente em todos os subgrafos decompostos. Se cada subgrafo é colorido com k cores, o grafo inteiro também pode ser colorido com k cores. Os subgrafos resultantes das partições correspondem a segmentos do código do programa. Desta maneira, o programa pode ser particionado apenas examinando-se seu código, não sendo necessária a construção de seu grafo de interferência inteiro. A combinação das colorações dos subgrafos também envolve a renomeação de cores em um dos

subgrafos para que todos utilizem a mesma cor para os nodos presentes no separador de *cliques* (que está presente em todos os subgrafos decompostos por ele).

No método de Gupta, Soffa e Steele, o *live range* é definido como sendo um grupo isolado de instruções de código contínuas ao invés de blocos básicos inteiros, nos quais a variável é definida e referenciada. Desta forma, a variável pode ter muitos *live ranges* em um único bloco básico. Os nodos de um grafo de interferência de um segmento de código correspondem aos *live ranges* ou *spans* de uma variável. Os *spans* que representam valores que estão simultaneamente vivos em algum ponto do programa são conectados por uma aresta, não podendo ser coloridos com a mesma cor.

Os *spans* que formam um separador de *cliques* estão presentes nos grafos de interferência das partições do código que precedem e que sucedem o *clique*, podendo ser atribuídos a diferentes cores em cada um dos grafos. Para resolver este problema e manter a semântica do programa, em código sem desvios, os registradores de um dos segmentos podem ser renomeados para que o mesmo registrador seja usado em todos os segmentos. Já em códigos com presença de desvios, as partições que sucedem e precedem um segmento de código podem ter atribuído registradores que não podem ser renomeados. Nesta situação, código para transferir valores de um registrador para outro pode ser inserido, ou seja, instruções de cópia (*move*) de registradores são inseridas. Na coloração, também pode haver casos de derramamento e de divisão de *live ranges*, como aqueles existentes nos outros métodos de alocação de registradores. Se o número de registradores disponíveis for menor que o número de *live ranges*, o algoritmo deve escolher valores para serem derramados para a memória e para serem atribuídos a registradores. Um *live range* também pode ser dividido, sendo que cada subdivisão pode ser atribuída a um registrador diferente, sendo que instruções de cópia de registradores devem ser inseridas. As decisões de derramamento são baseadas nas prioridades do nodo.

A prioridade de um nodo é medida pela economia no tempo de execução obtida se a variável correspondente ao nodo for atribuída a um registrador. A economia no tempo de execução (*NETSAV*) se um *live range* for atribuído a um registrador é dada por:

$$NETSAV = LODSAV * u + STRSAV * d - MOVCOST * n$$

sendo *LODSAV* a economia no tempo de execução se cada uso da variável estiver em um registrador e não na memória, *STRSAV* a economia se toda definição da variável estiver em um registrador e não na memória, e *MOVCOST* é o custo de se mover um valor entre um registrador e a memória. A variável u representa o número de usos, d o número de definições, e n é o número de *loads* e *stores* que devem existir para que o valor esteja no registrador durante o período pelo qual ele deve ser alocado a um registrador ou à memória.

O *live range* de uma variável pode se estender a vários blocos básicos. Uma variável referenciada dentro de um laço tem mais probabilidade de ser referenciada, logo se ela for atribuída a um registrador, a economia resultante desta alocação provavelmente é maior. Portanto, a economia total gerada pela alocação de um *live range* a um registrador é normalizada em relação a profundidade do aninhamento do laço, sendo dada por:

$$TOTALSAV = \sum_{i \in lr} (NETSAV\, V_i * w_i)$$

onde w_i é a profundidade do aninhamento do laço do bloco básico i e $NETSAV$ é a economia gerada pela atribuição da variável a um registrador no bloco básico i. Nota-se que, no método de Chaitin, o custo de derramamento de uma *web* também é calculado em relação ao aninhamento do laço.

Resumindo, no algoritmo de Gupta, Soffa e Steele, primeiro o programa é particionado em segmentos de código, sendo um grafo de interferência construído e colorido para cada partição. As prioridades de cada nodo são computadas globalmente e recomputadas depois de uma divisão de *live range* similarmente como é feito na coloração baseada em prioridade [8]. Durante a coloração de uma partição, somente as prioridades dos nodos que pertencem àquela partição são necessárias, reduzindo o tempo de execução do algoritmo. Os subgrafos são construídos e coloridos um a um. Após a coloração de um subgrafo, suas cores podem ser renomeadas para combinar com as cores de partições adjacentes. Se algum código de cópia for necessário, ele é introduzido. Os nodos no grafo que possuem menos vizinhos que o número de registradores disponíveis são coloridos por último, já que eles podem ser coloridos independentemente das cores atribuídas a seus vizinhos. Esse nodos são chamados de nodos sem restrições *(unconstrained)*, enquanto os nodos restantes são chamados de nodos com restrições *(constrained)*. Somente os *spans* com $TOTALSAV$ positivo são atribuídos a registradores.

O método é eficiente, gera código de boa qualidade, porém, como para fazer a alocação de cada segmento de código, é usada a coloração de grafos, o método de Gupta, Soffa e Steele possui os mesmos problemas que o método de Chaitin.

2.9 Método de George e Appel

Instruções de *move* ou cópia redundantes podem ser facilmente eliminadas com um grafo de interferência. Se não existir uma aresta no grafo de interferência entre o alvo e a fonte de uma instrução de cópia, a instrução pode ser eliminada. Os nodos alvo e fonte da instrução são combinados em um único novo nodo, cujas arestas são a união das arestas dos nodos combinados. Esta transformação é chamada de Combinação de Registradores *(Register Coalesce)*.

Chaitin combinava qualquer par de nodos não conectados por uma aresta em seu algoritmo de alocação de registradores [7]. Esta forma agressiva de combinação de registradores é muito eficiente na eliminação de instruções de cópia, porém, geralmente o novo nodo formado pela combinação dos nodos fonte e alvo da cópia possui mais arestas que os antigos, pois o número de arestas do novo nodo é a união das arestas dos nodos combinados. A consequência desta combinação de registradores agressiva é que, em certos casos, um grafo que era colorido com k cores antes do *coalesce*, não pode ser mais colorido com k cores após esta transformação.

Para resolver este problema, Briggs desenvolveu uma estratégia de combinação de registradores conservativa [4]. Nesta estratégia, se um nodo sendo combinado possui menos que k vizinhos de nível significante (nodos com k ou mais vizinhos), é garantido que a transformação de combinação de registradores não faz um grafo colorido com k cores antes da aplicação da transformação não ser mais colorido com k cores após a sua aplicação. Esta abordagem conservativa é eficiente em relação à eliminação de instruções de cópia sem a introdução de derramamento, porém ela não elimina todas as cópias redundantes, algumas ainda permanecem no código. Ele usa então uma

heurística de coloração *biased* durante a fase de seleção das cores: ao se colorir um temporário X que está envolvido em uma instrução de cópia $X \leftarrow Y$ ou $Y \leftarrow X$, se Y já tiver sido colorido, a cor de Y é selecionada para X, se possível. No caso de Y ainda não ter sido colorido, uma cor para X é selecionada de modo que possa ser a mesma cor de Y. Se X e Y possuírem a mesma cor, a instrução de cópia não é necessária. Mesmo assim, a abordagem de Briggs continua sendo falha, pois é fraca ao se lidar com um número grande de *moves*.

Resumindo, a transformação de Combinação de Registradores do método de alocação de registradores de Chaitin [7] produz muitos derramamentos, enquanto que a combinação de registradores conservativa do método de Briggs [4] deixa muitas instruções de *move* desnecessárias no código.

George e Appel [14] desenvolveram uma nova abordagem para alocação de registradores que entrelaça a fase de simplificação do algoritmo de Chaitin com a combinação de registradores conservativa de Briggs, produzindo uma abordagem que elimina mais instruções de cópia desnecessárias que a abordagem de Briggs com a garantia de que nenhum derramamento é introduzido no código. Nesta abordagem, o passo de combinação de registradores e de simplificação são chamados em um laço, sendo que a fase de simplificação é executada primeiro. Já na abordagem de Chaitin e de Briggs, a combinação de registradores é executada antes da simplificação, não existindo laço entre elas.

Existe um teorema [14] que diz que se um grafo de interferência G pode ser colorido utilizando a heurística de simplificação, a aplicação da transformação de combinação de registradores em um grafo intermediário produzido após alguns passos de simplificação de G, produz um grafo que também pode ser colorido. Isto garante que nenhum derramamento é introduzido pela abordagem desenvolvida por George e Appel. A técnica desenvolvida por eles é interessante, pois o primeiro passo de simplificação remove uma porcentagem tão grande de nodos, que faz com que a fase de combinação de registradores seja aplicada a todas as instruções de *move* apenas uma única vez.

Existem cinco fases principais no método de George e Appel, que pode ser chamado de Combinação de Registradores Iterada (*Iterated Register Coalescing*):

1. Construção (*Build*): constrói o grafo de interferência e identifica os nodos como sendo relacionados a *move* ou não, sendo um nodo relacionado a *move* um nodo que é a fonte ou o alvo de uma instrução de *move*;

2. Simplificação (*Simplify*): os nodos não relacionados a *move* que possuem nível menor são removidos do grafo, um por um;

3. Combinação (*Coalesce*): efetua a combinação de registradores conservativa de Briggs no grafo reduzido obtido pela fase de simplificação. A estratégia conservativa tem uma probabilidade maior de encontrar *moves* para serem combinados do que encontraria no grafo inicial, já que a fase de simplificação reduz o nível de muitos nodos. Após a combinação de dois nodos, se o nodo resultante não for mais relacionado a *move* ele fica disponível para a próxima rodada da simplificação, sendo marcado como não relacionado a *move*. As fases de Simplificação e de Combinação de Registradores são repetidas até que fiquem no grafo apenas nodos de nível alto ou relacionados a *move*;

4. Congelação (*Freeze*): se as fases de simplificação e de combinação de registradores não se aplicarem, nodos relacionados a *move* com nível menor são procurados e os *moves* nos quais esses nodos estão envolvidos são congelados. Isto significa que não há mais esperanças de que esses nodos sejam combinados, e eles são considerados não relacionados a *move*. As fases de simplificação e de combinação de registradores podem ser efetuadas novamente;

5. Seleção (*Select*): cores são selecionadas para cada nodo.

O método de George e Appel é compatível tanto com a coloração pessimista de Chaitin quanto com a coloração otimista de Briggs.

Os resultados desta abordagem foram comparados com os resultados da abordagem de Briggs. O número de derramamentos foi idêntico nas duas abordagens, como era o esperado, porém o número de instruções de cópia desnecessárias removidas foi muito maior na abordagem de George e Appel.

Uma abordagem de alocação de registradores que alterna as fases de simplificação e de combinação de registradores elimina muito mais instruções de *move* do que as abordagens anteriores, que efetuam a fase de combinação de registradores antes da simplificação. Esta abordagem é de simples implementação e fácil incorporação nas abordagens de coloração de grafos existentes.

Assim como o método de Briggs [4], o método de George e Appel é uma extensão do método de Chaitin, possuindo seus defeitos: inexistência de análise de fluxo de controle, fluxo de dados e custos de derramamento calculados localmente em um único bloco básico.

2.10 Conclusões

Pode-se observar que, em sua maioria, todas as abordagens globais para alocação de registradores são baseadas na coloração de grafos, carregando, portanto, seus defeitos. Este é o motivo pelo qual outras técnicas de alocação vêm sendo pesquisadas.

O trabalho de dissertação descrito neste texto tenta suprir os principais defeitos do método de coloração de grafos. A técnica implementada nesta dissertação faz análise de fluxo de controle do programa, faz análise de fluxo de dados e não utiliza a coloração de grafos para se fazer a alocação, utilizando uma abordagem completamente diferente. Esta abordagem é descrita na Seção 3 e foi proposta por Ottoni e Araújo [12].

Capítulo 3

Alocação Baseada em *Live Range Growth*

3.1 Introdução

A solução baseada em Crescimento de Domínios Ativos foi proposta por Ottoni e Araújo [12] para processadores dedicados DSPs *(Digital Signal Processors)* com o objetivo de resolver o problema da alocação de referências a vetores em registradores de endereçamento dentro de laços que contêm mais de um bloco básico. Nossa proposta é utilizar este método para a alocação global de registradores em processadores de propósito geral. Para facilitar o entendimento de nossa implementação e nossa contribuição, esta seção descreve em detalhes o método para Alocação Global de Registradores Baseado em Crescimento de Domínios Ativos, proposto por Ottoni e Araújo [12].

3.2 Método de Ottoni e Araújo

O problema central para se obter uma solução baseada em Crescimento de Domínios Ativos (*Live Range Growth*) para a Alocação Global de Registradores é o cálculo do menor custo associado a um dado domínio ativo. Neste algoritmo, um domínio ativo (*live range*) é definido como um conjunto de variáveis que são atribuídas a um mesmo registrador. Logo, todas as variáveis de um mesmo domínio ativo são alocadas ao mesmo registrador, fazendo com que todos os usos e definições destas variáveis, *webs*, sejam feitos via este registrador. Inicialmente, cada *web* [18] é colocada separadamente em um domínio ativo. A seguir, é usado um algoritmo heurístico, denominado Crescimento de Domínios Ativos, o qual faz uma junção sucessiva de pares de domínios ativos, até que o número total deles atinja o número de registradores disponíveis na arquitetura em questão. Para decidir o par de domínios ativos unidos a cada iteração do algoritmo, todas as combinações de pares de domínios ativos são avaliadas, e a que resulta em um menor custo é escolhida. O custo de um domínio ativo é medido pelo número de instruções de *load* e *store* necessárias para o ajuste do registrador. Assim, o problema central desta técnica é a determinação, para um dado domínio ativo, do número destas instruções para manter o registrador com a variável

correta durante todo o fluxo de execução do programa.

Neste método, a propriedade de que cada referência[1] deve ser precedida por uma única outra referência a uma variável que pertença ao mesmo domínio ativo é muito importante. Com esta propriedade determina-se, em cada ponto do programa, independentemente do caminho executado, qual variável será atribuída ao registrador alocado às variáveis deste domínio ativo. Para satisfazer esta propriedade, o programa é transformado em uma representação baseada em uma forma denominada *Single Reference Form (SRF)*. Esta forma foi proposta por Cintra e Araújo [10] e é uma adaptação da forma SSA (*Static Single Assignment*) [11], possuindo a propriedade de que cada referência pode somente ser precedida por uma única outra referência. Neste problema, não é importante o fato de uma referência ser sucedida por uma única outra referência a uma variável de seu domínio ativo. Quando uma instrução é executada, o registrador certamente está alocado ao domínio ativo especificado na instrução, sendo ele um uso ou uma definição da variável em questão. Outro conceito importante para entender o funcionamento desta técnica está relacionado com a função ϕ. Função ϕ é uma forma especial de definição que recebe como argumento um conjunto de definições $\{x_1, ..., x_n\}$ de uma variável x que atingem um ponto de junção no Grafo de Fluxo de Controle (CFG) do programa, um vértice no CFG com mais de um predecessor, e produz uma nova definição x_{n+1} para a variável x [10]. Assim sendo, diz-se que $x_{n+1} = \phi\ (x_1, ...\ , x_n)$. Ou seja, os argumentos da função ϕ são todas as variáveis pertencentes aos domínios ativos unidos que chegam até a função, que é resolvida de modo a escolher a variável dentre os argumentos que estará armazenada no registrador alocado aos domínios ativos a partir deste ponto no programa. Para ilustrar este conceito, suponha o trecho de código do programa mostrado na Figura 1, onde as variáveis consideradas para a união são b e i. A função ϕ que pertence ao bloco básico 6 tem como argumentos as variáveis b e i, que são as variáveis consideradas para a união, e define qual dos argumentos está alocado ao registrador a partir dela.

[1] referência é definida como o uso ou definição de uma variável

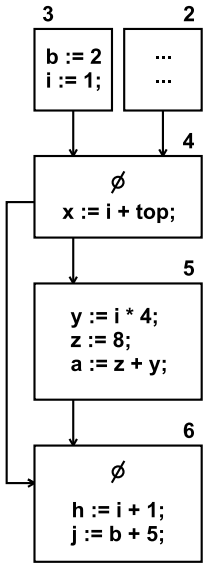

Figura 1: Exemplo de trecho de programa a ser submetido ao método de Alocação baseado em Crescimento de Domínios Ativos

O primeiro passo do algoritmo de Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos é a inserção de funções ϕ na entrada de cada bloco básico que pertença à fronteira de dominância iterada[2] dos blocos básicos que usam ou definem alguma variável do domínio ativo. Isto é necessário porque tanto um uso quanto uma definição de uma variável obrigam que a mesma esteja alocada ao registrador. Na modelagem proposta por Ottoni e Araújo, duas informações são essenciais:

1. a determinação de qual variável do domínio ativo está atribuída ao registrador associado a este domínio ativo em cada ponto do programa;

2. determinar se a variável atribuída ao registrador possui valor igual ou diferente do seu valor armazenado na memória. Quando os valores são iguais, diz-se que o valor está consistente, senão inconsistente. Esta informação é importante, pois economiza-se uma instrução de *store* se os valores contidos no registrador e na memória forem iguais, no caso de precisar usar este registrador para outra variável do domínio ativo em um determinado ponto do programa.

O segundo passo do algoritmo é o cálculo das informações descritas no parágrafo anterior. Para se calcular qual variável estará alocada ao registrador de um domínio ativo em cada ponto do programa, juntamente com seu estado de consistência, usa-se a análise de fluxo de dados, chamada Análise de Alcançabilidade e Consistência de Registradores (*Register Consistency Reachability, (RCR)*).

[2]fronteira de dominância iterada são os blocos que possuem mais de um antecessor ou o último bloco

Seja V o conjunto das variáveis que pertencem ao domínio ativo, C o estado de consistência com a memória, I o estado de inconsistência, e X o desconhecimento se o valor armazenado no registrador está consistente ou não com a memória. Define-se também os conjuntos $V_\phi = \{\omega_i\}$ e $E_\phi = \{\sigma_i\}$. Note que o resultado de uma função ϕ precisa também ser formado por um par: uma variável de V e um estado de consistência C ou I. Porém, para se conhecer as soluções das funções ϕ, é necessário a obtenção dos resultados da análise RCR para identificar quais estados de registrador alcançam a função ϕ e avaliar qual a melhor solução para ela. A solução de uma função ϕ_i é composta pelo par (ω_i, σ_i). Os ω_is denotam as variáveis e σ_is denotam os estados de consistência do problema de otimização que tentam diminuir o número de instruções de *load* e *store*. Assim, a fim de minimizar o custo do domínio ativo, deseja-se escolher as soluções das funções ϕ, ou seja, a solução da função ϕ será a variável e o estado de consistência que fizerem com que menos instruções de acesso à memória sejam inseridas no código.

Para esta análise RCR, os itens são pares *(v, e)* onde v é uma variável do domínio ativo e e é um estado de consistência, nos quais:

- $v \in \text{V} \cup V_\phi \cup \{\epsilon\}$, onde ϵ significa que nenhuma variável está no registrador;
- $e \in \{\text{C,I,X}\} \cup E_\phi$.

Um conjunto de itens é calculado para cada ponto entre duas referências do programa, para que se saiba qual variável está alocada ao registrador depois de cada referência a uma variável do domínio ativo. Por exemplo, em uma instrução do tipo x := y + z, sendo x e y variáveis do domínio ativo, um ponto de cálculo é considerado antes da leitura de y, outro entre a leitura de y e a escrita em x, e outro ponto após a escrita em x. O uso de z é ignorado no cálculo do item relativo a este domínio ativo, pois ele não pertence ao domínio ativo. Para cada referência r, o conjunto de elementos de RCR que atinge sua entrada é dado por todos os elementos que atingem a saída de alguma referência p que pode preceder r no fluxo de controle do programa, ou seja, $\text{in[r]} = \bigcup_{p\ pred\ r} \text{out[p]}$. Define-se que a primeira referência do programa é vazia e seu estado de consistência é X, formalmente, $\text{in}[r_o] = \{(\epsilon,\text{X})\}$. Ressalta-se que em todos os pontos que precedem uma referência, o conjunto de itens do RCR somente pode conter um único elemento do mesmo domínio ativo. Isto é o resultado da inserção das funções ϕ, e é similar a propriedade de um programa em SSA, onde cada uso de uma variável é alcançado por uma única definição desta variável.

Segundo Ottoni e Araújo [12], existem três casos, dependendo do tipo de referência: uso, definição ou função ϕ, para a obtenção dos elementos que alcançam a saída de uma referência r. Seja LR o domínio ativo em questão e s um estado de consistência qualquer:

- *Caso 1*: r: x:= ... | x ∈ LR
  out[r] = {(x, I)};
- *Caso 2*: r: ... := x | x ∈ LR

$$out[r] = \begin{cases} \{(x,s)\}, & \text{if } in[r] = \{(x,s)\}, \forall s \\ \{(x,C)\}, & \text{if } in[r] = \{(y,s)\}, y \in (V \cup \{\epsilon\}) - \{x\}, \forall s \\ \{(x,X)\}, & \text{if } in[r] = \{(\omega_i,\sigma_i)\}, \omega_i \in V_\phi \end{cases}$$

- *Caso 3*: r: ϕ_i
  out[r] = $\{(\omega_i, \sigma_i)\}$

No Caso 1 existe uma definição da variável x. Logo, após esta referência o registrador contém a variável x com um valor inconsistente com o seu valor presente na memória, pois um novo valor acabou de lhe ser atribuído.

O Caso 2 se subdivide em três casos, dependendo do item que precede a referência. No primeiro subcaso in[r] é constituído de um estado no qual x já estava no registrador antes da referência em questão. Logo, um uso de x faz com que o registrador continue com x no mesmo estado de consistência de antes. O segundo subcaso é quando in[r] possui um estado com uma outra variável y, que pertence ao mesmo domínio ativo que x, no registrador ou o registrador vazio. Neste caso, independentemente da consistência de y, é necessário um *load* de x antes da referência r, fazendo com que o registrador agora contenha x com um valor consistente ao da memória, pois o mesmo terá de ser carregado da memória. Já o terceiro subcaso trata a situação de a referência r ser atingida por uma função ϕ. Neste caso, garante-se apenas que o registrador contém o valor de x, contudo o seu estado de consistência é indeterminado e depende da solução escolhida para a função ϕ. Esta solução pode ser por exemplo o próprio x.

O Caso 3 trata das funções ϕ, consideradas como um tipo especial de referência, que deixa o registrador em um estado dado pela solução escolhida para ela.

Expressões para os conjuntos *in* e *out* podem ser usadas para se encontrar iterativamente estes conjuntos para todos os pontos do programa, assim como são resolvidas as expressões das outras análises de fluxo de dados, como, por exemplo, a de variáveis vivasc̃itemuchnick.

No terceiro passo do algoritmo, valendo-se da propriedade de que, após a inserção das funções ϕ, toda referência a alguma variável do domínio ativo é alcançada por um único estado de registrador, o conjunto mínimo de instruções necessárias para o ajuste do conteúdo do registrador pode ser calculado para cada referência que não depende da solução de funções ϕ.

Sendo $live_{in}$[r] o conjunto de variáveis vivas no ponto antes da referência r, os seguintes casos definidos por Ottoni e Araújo [12] identificam as instruções que não dependem da solução de funções ϕ:

1. *Caso 1*: r: x:= ... | x ∈ LR, in[r] = {(v, e)}
   (a) Caso (v ∈ V - {x}) e (e = I) e (v ∈ $live_{in}$[r]):
   a instrução necessária antes de r é *store* v. O registrador contém uma variável v diferente de x, que está inconsistente e viva neste ponto do programa, logo seu valor deve ser armazenado. Como a referência a x é uma definição, seu valor não precisa ser carregado da memória.
   (b) Caso contrário, nenhuma instrução é necessária, ou sua necessidade depende da solução de alguma função ϕ.
2. *Caso 2*: r: ... := x | x ∈ LR, in[r] = {(v, e)}
   (a) Caso v = ϵ:

a instrução necessária antes de r é um *load* x. O registrador não contém nenhuma variável, e como a referência a x é um uso, é necessário carregar o valor de x da memória.

(b) Caso (v $\in$ V - {x}) e ((e $\in$ {C,X}) ou ((e=I) e (v $\notin$ live$_{in}$[r]))):
a instrução necessária antes de r é um *load* x. O registrador contém uma variável v diferente de x, que está consistente ou possui estado de consistência indefinido, ou está inconsistente mas não está viva neste ponto. Dessa forma, não é necessário armazenar o valor de v. Como a referência a x é um uso, seu valor deve ser carregado da memória.

(c) Caso (v $\in$ V - {x}) e (e=I) e (v $\in$ live$_{in}$[r]):
instrução necessária é um *store* v; *load* x. O registrador contém uma variável v diferente de x, que está inconsistente e viva neste ponto do programa, logo o valor de v deve ser armazenado. E como a referência a x é um uso, seu valor deve ser carregado da memória.

(d) Caso contrário, nenhuma instrução é necessária, ou sua necessidade depende da solução de alguma função ϕ.

O passo final do algoritmo trata da solução das funções ϕ.

Após a inserção das instruções de *load* e *store* que não dependem de funções ϕ, as instruções dependentes de soluções das funções ϕ devem ser inseridas. Além desta inserção, as funções ϕ devem ser resolvidas, ou seja, decidir qual variável é atribuída ao registrador e seu estado de consistência em cada um dos pontos do programa onde uma função ϕ foi inserida.

Dada uma função $\phi_i = (\omega_i, \sigma_i)$, para se calcular o custo de uma solução (ω_i, σ_i) = (w,f), analisa-se todas as referências do programa que:

A1. são alcançadas por ϕ_i, ou seja, possuem (ω_i, σ_i) no seu conjunto in; ou

A2. são alcançadas pelo registrador com uma variável em estado indefinido devido à solução de ϕ_i; ou

A3. alcançam ϕ_i.

Para solucionar os Itens A1 e A2, propaga-se a solução (w,f) da função ϕ_i como é feito na análise de fluxo de dados RCR, e usa-se a análise de casos descrita a para emissão de instruções não dependentes de funções ϕ, já que os estados de registrador na entrada destas referências estão bem determinados.

As referências do Item A3 são as contidas em in[ϕ_i]. Logo, segundo [12], para avaliar o custo de ϕ_i relacionado a cada uma destas referências r, deve-se comparar a solução de ϕ com cada (v,e) pertencente a in[ϕ] fazendo a seguinte análise de casos:

B1. (w = ϵ) e (v $\in$ V) e (e=I) e (v $\in$ live$_{in}$[ϕ_i]):

instrução necessária após r é *store* v. A solução de ϕ_i é não ter variável alguma usando o registrador. Uma variável v que esteja viva e inconsistente no registrador deve ser armazenada na memória.

B2. (w $\neq \epsilon$) e (v = w) e (f=C) e (e=I):

instrução necessária após r é *store* v. O registrador contém a variável necessária, porém em estado inconsistente. Como a solução demanda que a variável esteja consistente, uma instrução de *store* é necessária.

B3. (w $\neq \epsilon$) e ((v=ϵ) ou ((v $\in$ V - {w}) e ((e=C) ou (v $\notin$ live$_{in}[\phi_i]$))):

instrução necessária após r é *load* w. O registrador não contém variável alguma, ou contém uma variável v diferente da solução w, mas v está consistente ou não está viva neste ponto. Portanto, não é necessário armazenar v na memória, apenas tem-se que carregar w. Mesmo que f=I, o resultado desta instrução será v consistente com a memória sem custo adicional para isto.

B4. (w $\neq \epsilon$) e (v $\in$ V -{w}) e (e=I) e (v $\in$ live$_{in}[\phi_i]$):

instruções necessárias após r são *store* v; *load* w. O registrador contém uma variável v diferente da solução w, e v está inconsistente e viva neste ponto. Portanto é preciso armazenar v na memória, e carregar w. Mesmo que f=I, o resultado destas instruções terá v consistente com a memória, sem custo adicional para isto.

B5. Caso contrário, nenhuma instrução é necessária.

Pode existir uma situação em que o valor de uma função ϕ_i dependa de outra função ϕ_j. Para resolver uma função ϕ neste caso, pode-se usar uma estratégia gulosa, que ordena as funções ϕ de acordo com algum critério, e então as resolve seguindo esta ordenação. Para cada função ϕ, tenta-se todas as suas possíveis soluções e escolhe a que resultar no menor custo. Para calcular os custos de uma dada função ϕ_i, pode-se usar as soluções de outras funções ϕ já resolvidas das quais ϕ_i depende. Aquelas outras funções ϕ das quais ϕ_i depende, mas que ainda não foram solucionadas são ignoradas.

3.3 Exemplo

Para ilustrar o método de Ottoni e Araújo, a Figura 1 mostra um trecho de código, na forma de grafo de fluxo de controle, onde cada retângulo representa um bloco básico. Um bloco básico é definido como uma sequência de zero ou mais instruções, onde nenhuma delas é uma instrução de desvio de controle [25].

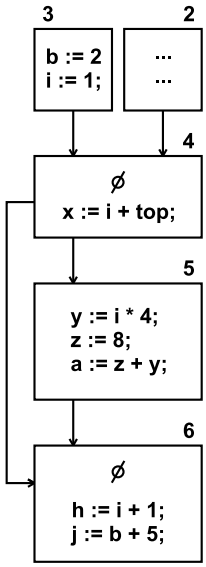

Figura 1: Exemplo de trecho de programa a ser submetido ao método de Alocação baseado em Crescimento de Domínios Ativos

Na iteração mostrada como exemplo, deseja-se unir os domínios ativos b e i. Uma função ϕ foi inserida no Bloco Básico 6 pois este bloco possui dois predecessores: os blocos 4 e 5. Já outra função ϕ foi inserida no Bloco Básico 4 pois este também possui dois predecessores: os blocos 2 e 3. O primeiro passo do algoritmo, após a construção dos domínios ativos e da inserção de funções ϕ é a Análise de Alcançabilidade e Consistência de Registradores. Fazendo esta análise apenas para os domínios ativos a serem unidos (b e i), obtem-se os itens mostrados na Figura 2.

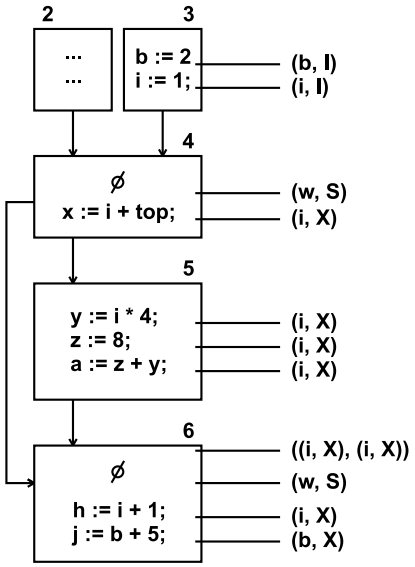

Figura 2: Trecho do programa com os respectivos itens gerado pela Análise de Alcançabilidade e Consistência de Registradores

O segundo passo do algoritmo é a emissão de instruções não dependentes de função ϕ. No Bloco Básico 3, a primeira instrução é de definição de b, já a segunda é de definição de i. Como b e i são unidos, logo ocupam o mesmo registrador. Antes da segunda instrução, de definição de i, pela análise RCR, o registrador alocado à união de b e i contém a variável b, (logicamente diferente de x), em um estado inconsistente com a memória e viva neste ponto do programa, pois é utilizada na última instrução do Bloco Básico 6. Logo, tem-se o Caso 1.a da análise de casos apresentada para a emissão de instruções não dependentes das funções ϕ, que demanda uma instrução de *store* b antes da instrução de definição de x. Já no último bloco básico, o bloco de número 6, em sua última instrução, a variável b é utilizada. Porém, pela análise RCR, o registrador contém a variável x em um estado X, o qual depende da solução da função ϕ. Como x não está mais viva neste ponto do programa, tem-se o Caso 2.b da análise de casos da emissão de instruções não dependentes das funções ϕ, o que demanda um *load* b antes da instrução que o utiliza.

O último passo do algoritmo é a construção do grafo de dependências ϕ para os domínios ativos a serem unidos. O grafo tem a forma mostrada na Figura 3. Existe uma aresta que liga a função ϕ_1, presente no Bloco Básico 4 com a função ϕ_2, presente no Bloco Básico 6. Esta aresta existe porque a função ϕ_2 depende da solução da função ϕ_1, pois no item da RCR que alcança ϕ_2, i está em um estado de consistência X, que depende da solução escolhida para a função ϕ_1.

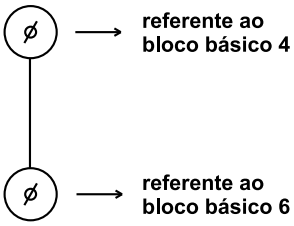

Figura 3: Grafo de Dependências ϕ (DG_ϕ) para os domínios ativos b e i

Para a solução da função ϕ do Bloco Básico 4, é preciso analisar as referências do programa que são alcançadas por ϕ, são alcançadas pelo registrador com uma variável em estado indefinido por causa da solução de ϕ e as que alcançam ϕ. No primeiro caso, Item A1, a referência alcançada por ϕ é i na instrução $x := i + top$. No segundo caso, Item A2, todas as referências a i no próprio Bloco 4, assim como nos Blocos 5 e 6 são alcançadas pelo registrador com uma variável, no caso i, em um estado indefinido (X) devido a solução de ϕ. Já no terceiro caso, Item A3, quem alcança ϕ é o item *(i, I)*. Caso a solução de ϕ seja não ter variável alguma utilizando o registrador, como o item que alcança ϕ é *(i, I)* e i está viva neste ponto do programa (é usada no Bloco Básico 5 e no 6), tem-se um custo de um *store* i no Bloco Básico 3 e um *load* i no Bloco Básico 4, pois i será usada na instrução seguinte a ϕ. Se a solução for i em um estado C, consistente com a memória, tem-se um custo de um *store* i em 3, para que i fique consistente com a memória. Se a solução for b em um estado de inconsistência qualquer, como b e i estão vivas neste ponto do programa, tem-se um custo de um *store* i e um *load* b no Bloco Básico 3. Entretanto, se a solução for i em um estado inconsistente, tem-se um custo zero. Logo, esta última solução é a escolhida, pois tem custo 0.

Resolvendo-se a função ϕ do Bloco Básico 6, tem-se que esta função ϕ depende da função ϕ_1, do bloco básico 4, que já foi resolvida como *(i, I)*. Logo esta é a referência que alcança ϕ. A referência que é alcançada por ϕ é um uso de i na instrução $h := i + 1$. As referências que são alcançadas pelo registrador com uma variável em estado indefinido devido a solução de ϕ é um uso de b na instrução seguinte. Fazendo uma análise de casos para a escolha da solução de ϕ, tem-se um quadro semelhante ao da solução da função ϕ anterior, logo a solução da função ϕ é *(i, I)*, que possui custo de 0. Alocando-se o registrador r à união de b e i, com as instruções já emitidas, obtem-se o código mostrado na Figura 4.

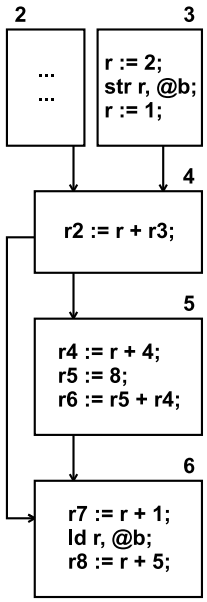

Figura 4: Código resultante da união dos domínios ativos b e i

3.4 Conclusão

Esta seção descreveu em detalhes a descrição do método de Alocação Global de Registradores Baseado em Crescimento de Domínios Ativos, desenvolvido por Ottoni e Araújo [12].

Segundo [12], a complexidade do problema de resolver as funções ϕ de forma ótima faz com que este seja um problema NP-completo, não existindo, portanto, um método geral, eficiente e ótimo, para resolver estas funções. Entretanto, Ottoni e Araújo sugerem como outra alternativa o uso de um grafo de dependência ϕ (DG_ϕ) para se calcular as dependências entre as funções ϕ. O grafo DG_ϕ é definido em [12] como um grafo não dirigido no qual existe um vértice associado a cada função ϕ, e existe uma aresta (w_i, w_j), entre dois vértices w_i e w_j, se e somente se, w_i é uma função ϕ de w_j e vice-versa. Caso este grafo seja acíclico, o algoritmo LRO *(Leaves Removal Order)* [12] pode ser usado para determinar de forma eficiente e ótima a solução para todas as funções ϕ.

Este método de alocação foi apenas proposto em [12], não tendo sido implementado. Logo, a sua eficiência na prática, ainda não havia sido avaliada, o que foi feito neste trabalho de dissertação.

Capítulo 4

Alocação de Registradores Baseado em Crescimento de Domínio Ativo e Combinação de Registradores

4.1 Introdução

Esta seção apresenta as implementações do algoritmo de Alocação Global de Registradores Baseado em Crescimento de Domínios Ativos proposto por Ottoni e Araújo. Ela está organizada da seguinte forma: após a descrição das estruturas de dados utilizadas nas implementações, são mostrados nas Subseções 4.3 e 4.3.1, respectivamente, o algoritmo proposto para o método de Ottoni e Araújo, seguido da descrição de sua implementação.

Primeiramente foi implementado um algoritmo que, para a solução da função ϕ, utiliza a última análise de casos descrita na Seção 3.2 para analisar as referências alcançadas por ϕ. Para esta implementação, a fim de se economizar uma passada pelo código do programa, procura-se as referências nas instruções sucessoras da função ϕ que possuam um registrador com uma variável em estado indefinido (X) devido a solução de ϕ ou que possuam (ω_i, σ_i) em seu conjunto *in*.

As Subseções 4.4 e 4.4.1 apresentam, respectivamente, o algoritmo e a implementação do método de Ottoni e Araújo, acrescido da transformação de Combinação de Registradores proposta por nós e da implementação da solução descrita por Ottoni e Araújo para os itens A1 e A2 da Seção 3.2, que faz com que a análise do custo da resolução das funções ϕ se estenda a todos os blocos básicos do programa, e não apenas considere as instruções sucessoras à função ϕ.

Os resultados destas implementações são apresentados na Seção 5.

4.2 Estruturas de Dados

Uma cadeia-du (*du-chain*) [18] de uma variável conecta uma definição desta variável a todos os seus usos. No algoritmo implementado, os domínios ativos iniciais são as *webs* calculadas para o programa. Para se calcular as *webs*, primeiramente as cadeias-du são calculadas. A seguir, as *webs* são calculadas de acordo com o algoritmo descrito no livro de Muchnick [18]. Uma *web* para

uma variável é a união máxima da interseção das cadeias-du para essa variável. Para todas as cadeias-du encontradas, se elas tiverem símbolos iguais e pelo menos um uso igual, faz-se a união destas duas cadeias-du em uma *web*. Existe então nesta implementação, uma lista encadeada de estruturas de dados que contêm o símbolo de cada *web* e as instruções que a definem e a usam.

Também existe uma outra lista encadeada de estruturas de dados que representa os domínios ativos. A diferença entre um domínio ativo e uma *web*, é que o primeiro pode conter mais de uma *web*. Um domínio ativo é inicializado com uma *web*, assim, logo no início do algoritmo, um domínio ativo corresponde a uma *web*. À medida que o algoritmo é iterado e os domínios ativos unidos, um domínio ativo pode conter mais de uma *web*.

Observe que deve ser considerado o fato de que duas variáveis que são usadas simultaneamente não podem compartilhar o mesmo registrador, logo um domínio ativo que contém ambas não pode ser considerado para a união, já que pela definição de domínio ativo, ambas as variáveis compartilhariam o mesmo registrador ao mesmo tempo. Isto acontece em uma instrução cuja operação possui dois operandos. Por exemplo, $x + y = z$. Os operandos x e y não podem compartilhar o mesmo registrador, pois são usados simultaneamente. Entretanto, x e z ou y e z podem pertencer ao mesmo domínio ativo, compartilhando o mesmo registrador, pois quando z é definido, tanto x quanto y já foram usados. Para suprir esta restrição no algoritmo implementado, utiliza-se o conceito de interferência, bem parecido com interferência entre nodos no grafo de interferência do algoritmo de alocação de registradores baseado em coloração de grafos [7]. Neste último algoritmo, se dois nodos interferem entre si, eles não podem ser coloridos com a mesma cor. Em nosso algoritmo de alocação de registradores baseado em crescimento de domínios ativos, dois domínios ativos que interferem entre si não podem ser unidos. Assim, na estrutura de dados que representa um domínio ativo, é inserido um campo que guarda uma lista de domínios ativos que interferem com o domínio ativo representado. Quando dois domínios ativos são unidos, suas interferências também são unidas.

Uma outra consideração importante que deve ser armazenada na estrutura de dados de um domínio ativo é se ele deve ser alocado a um registrador que obedece à convenção de registradores salvos pelo módulo chamado (*callee-saved*), ou seja, registradores que devem ser restaurados pelo procedimento chamado quando este os altera. Uma análise de variáveis vivas é feita, e um candidato à alocação que está vivo entre chamadas de rotinas deve ser forçado a ficar em um registrador salvo pelo módulo chamado (*callee-saved*). Logo, a estrutura de dados de um domínio ativo também possui um campo que informa se o domínio ativo deve ser alocado a um registrador deste tipo.

4.3 Algoritmo de Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos

O algoritmo para a alocação global de registradores baseado em crescimento de domínios ativos, em linguagem de alto nível, pode ser descrito da seguinte forma:

```
make_webs;
live_ranges = webs;
insert_phis;
make_interferences;
create_DGgraph;
while (numLiveRanges > numRegs) do
        make_rcr;
        emit_instructions_independents_phi;
        solution_phi_function;
substitute_liveRanges_regs;

procedure solution_phi_function()
        form_pairs;
        for each pair do
                if not(phi depends on another phi) then
                        case_analysis
                else
                        brute_force_heuristic;
                        case_analysis;
        cost = infinity;
        for each pair do
                if (pair->cost < cost) do
                        cost = pair->cost;
                        merge = cost;
        merge_pair(merge);
```

onde:

- `make_webs`:

  Função que percorre todo o código, procurando as *webs*;

- `insert_phis`:

  Função que calcula a fronteira de dominância iterada de cada domínio ativo (*live range*) encontrado e insere uma função ϕ relativa a este domínio ativo no início do bloco básico pertencente à fronteira de dominância iterada;

- `make_interferences`:

  Função que insere uma interferência entre domínios ativos que são operandos em uma mesma instrução. Esta interferência garante que estes domínios ativos não são unidos, impedindo-os de serem alocados a um mesmo registrador;

- `create_DGgraph:`

  Função que cria o grafo de dependência entre as funções ϕ;

- `make_rcr:`

  Função que faz a análise de fluxo de dados Alcançabilidade e Consistência de Registradores (*Register Consistency Reachabiliy*);

- `emit_instructions_independents_phi:`

  Função que calcula o conjunto mínimo de instruções para obter o conteúdo do registrador necessário para cada referência que não depende da solução de funções ϕ, valendo-se da propriedade de que, após a inserção das funções ϕ, toda referência a alguma variável do domínio ativo é alcançada por um único estado de registrador;

- `solution_phi_function:`

  Função que insere as instruções dependentes das soluções das funções ϕ;

- `substitute_liveRanges_regs:`

  Função que, ao final do algoritmo, substitui as referências: variáveis, registradores virtuais, temporários, pelos respectivos registradores alocados a elas;

- `form_pairs:`

  Esta função forma os pares de domínios ativos que terão o custo de sua união testado, desde que estes pares não interfiram entre si e cujas variáveis tenham o mesmo tipo;

- `case_analysis:`

  Dada uma função ϕ, para se avaliar o custo de uma solução, deve-se analisar todas as referências do programa que: (I) são alcançadas por ϕ, ou seja, possuem a solução de ϕ em seu conjunto in, (II) são alcançadas pelo registrador com uma variável em estado indefinido X por causa da solução de ϕ ou então (III) alcançam ϕ. Para resolver (I) e (II), a solução da função ϕ é propagada como na resolução da análise de fluxo de dados RCR e as instruções que agora não são dependentes de funções ϕ são emitidas. As referências de (III), que podem preceder a função ϕ no fluxo de execução, são as contidas em in[ϕ]. Esta função, então, faz uma análise de casos, comparando-se a solução de ϕ com cada *(v, e)* $\in$ *in[*ϕ*]* para avaliar o custo de ϕ relativo a cada uma das referências;

- `brute_force_heuristic:`

  Estratégia gulosa para se encontrar a solução de uma função ϕ_i que depende de outra função ϕ_j. A idéia básica desta técnica é ordenar as funções ϕ segundo algum critério e então resolvê-las seguindo esta ordenação. Para cada função ϕ, pode-se tentar todas as suas possíveis soluções, e escolher a que resultar em um menor custo. Para o cálculo destes custos, para uma dada função ϕ_i, pode-se utilizar as soluções para as outras funções ϕ_j já resolvidas das quais ϕ_i depende. Outras funções ϕ das quais ϕ_i depende, mas que ainda não foram resolvidas são ignoradas durante o cálculo da solução de ϕ_i;

- `merge_pair:`

  Função que, após comparar-se o custo de todas as uniões e escolher a de menor custo, efetua a união.

4.3.1 Implementação do Algoritmo

Inicialmente, percorre-se todo o código para identificar as *webs*, conforme o algoritmo descrito no livro de Muchnick [18], também identificando as interferências entre as variáveis usadas simultaneamente em uma mesma instrução. Os domínios ativos são inicializados com as *webs*.

Após a construção dos domínios ativos, funções ϕ devem ser inseridas na entrada de cada bloco básico que pertença à fronteira de dominância iterada dos blocos básicos que usam ou definem alguma variável do domínio ativo. Logo, a função ϕ relativa a este domínio ativo tem como argumentos as variáveis pertencentes ao domínio ativo. A fronteira de dominância de um nodo X em um grafo CFG é o conjunto de nodos Z, tal que X domina algum predecessor de Z, mas não todos [25]. Formalmente:

DF(X) = {Z | ∃ Y, Y′ ∈ PRED(Z)) [X DOM Y and X ¬DOM Y′]}

Para a inserção das funções ϕ, é utilizado o algoritmo *FUD ϕ-placement algorithm*, ou *Algorithm Placement*, ilustrado no livro de Michael Wolfe [25]. Primeiramente, os nodos no grafo CFG que possuem definições de cada variável devem ser identificados. Uma função ϕ é adicionada no último nodo do grafo CFG, o nodo de saída, para cada variável definida no programa.

Dados então:

- DF(X), a fronteira de dominância iterada no CFG de um nodo X;
- D(M), o conjunto de nodos CFG que contêm definições à variável M;
- o conjunto dos símbolos ou variáveis pertencentes ao programa;

o algoritmo insere uma função ϕ para cada variável pertencente ao conjunto de símbolos do programa.

Após a inserção de funções ϕ, é construído um grafo denominado Grafo de Dependência ϕ (DG_ϕ). Este grafo é indireto, nele existe um vértice para cada função ϕ inserida no programa, e existe uma aresta (w_i, w_j) entre dois vértices w_i e w_j, se e somente se, w_i é uma função ϕ de w_j e vice-versa. Para se construir este grafo, todos os nodos do grafo são pesquisados. Se uma função ϕ relativa a um domínio ativo tiver como entrada um estado de registrador que dependa da solução da função ϕ relativa a este mesmo domínio ativo presente em um nodo predecessor, normalmente com um estado de consistência X, a função ϕ do nodo depende da função ϕ do nodo predecessor. O grafo DG_ϕ é útil para se descobrir as dependências entre funções ϕ.

Enquanto o número de domínios ativos for maior que o número de registradores da máquina em questão, é feita a iteração do algoritmo. Inicialmente, é feita a Análise de Alcançabilidade e Consistência de Registradores (*Register Consistency Reachability*, ou *RCR*). Esta análise calcula a variável que estará alocada ao registrador de um domínio ativo em cada ponto do programa, bem como seu estado de consistência. Os itens da análise são calculados por iteração, até que os itens de uma iteração sejam iguais aos itens da iteração anterior. Como dito na Seção 3.1,

um conjunto de itens é calculado para cada ponto entre duas referências do programa, ou seja, antes e depois da leitura de cada operando da instrução e antes e depois da escrita do resultado da instrução. Para cada ponto do programa, o conjunto de elementos da análise que alcança sua entrada é dado por in[r] = $\bigcup_{p\ pred\ r}$ out[p]. E para se calcular os elementos para out[r] é feita uma análise de casos, como mostrado também na Seção 3.1, de acordo com o tipo de referência (uso ou definição) ou de se tratar de uma função ϕ, dependendo também do que está contido no elemento in[r]. Logo, para se computar os itens da análise, o grafo de fluxo de controle do programa é visitado em *Reverse Post Order*, sendo que, se o nodo tiver mais de um predecessor, o item que alcança sua entrada é a união de todos os itens *out* da última referência dos blocos básicos predecessores. A primeira referência do programa r_0 é inicializada com in[r_o] = {(ϵ,X)}.

Como apresentado no parágrafo anterior, a análise RCR é efetuada usando-se a abordagem Iterativa. Esta abordagem é a mais fácil de ser implementada e a mais utilizada entre as abordagens para se resolver problemas de fluxos de dados. Ela é chamada de iterativa porque uma coleção de equações de fluxo de dados é construída para representar o fluxo de informação e elas são solucionadas por iteração a partir de um conjunto de valores iniciais apropriados. Informações de fluxo de dados podem ser coletadas montando e resolvendo sistemas de equações que relacionam informações em vários pontos do programa.

O terceiro passo da iteração do algoritmo de alocação de registradores baseado em crescimento de domínios ativos é a emissão de instruções não dependentes de funções ϕ. Para tal, inicialmente é feita a análise de variáveis vivas do programa, e a seguir o grafo de fluxo de controle do programa é percorrido, e é aplicada a análise de casos descrita na Seção 3.1. Esta análise considera o tipo de referência, uso ou definição, o item que alcança a referência (in[r]), e o fato da variável presente em in[r] estar viva a partir deste ponto do programa ou não. As instruções necessárias para obter o conteúdo do registrador são então emitidas para cada referência que não depende da solução de funções ϕ. Uma referência não depende da solução de funções ϕ quando não possui {(ω_i, σ_i)} em in[r], ou então out[r] não contém nenhum item relativo a referência que tenha um estado de consistência indefinido (X) devido à solução de alguma função ϕ.

O último passo na iteração do algoritmo é a solução das funções ϕ. Neste passo, inicialmente escolhe-se os pares de domínios ativos a terem sua união testada. Para isso existe uma função que escolhe dois domínios ativos para serem unidos, desde que não interfiram entre si e sejam do mesmo tipo, por exemplo ponto flutuante ou inteiro. É dada a preferência para a escolha de um domínio ativo de cada caminho no grafo do fluxo de controle que chega até um nodo que contenha uma função ϕ. Por exemplo, no grafo de fluxo de controle mostrado na Figura 5, o Bloco Básico 4 contém a função ϕ. Este bloco possui dois blocos antecessores, logo possui também dois caminhos distintos que saem do bloco básico inicial e chegam até ele: um formado pelos Blocos 1 e 2, e o outro formado pelo Bloco 3. Desta forma, domínios ativos com referências pertencentes aos Blocos Básicos 1 e 2 são unidos com domínios ativos com referências pertencentes ao Bloco Básico 3. Esta escolha se deve ao fato de que, provavelmente, o custo da união de dois domínios ativos que pertencem a caminhos diferentes, logo bloco básicos diferentes, é menor. Além disso, menos instruções de ajuste do registrador são necessárias, pois a probabilidade dos domínios ativos pertencentes a blocos básicos diferentes estarem vivos ao mesmo tempo é menor. Quando não for possível a escolha de um domínio ativo pertencente a caminhos diferentes no grafo de fluxo do programa, todos pares com todas as combinações possíveis de domínios ativos são testados,

desde que possuam o mesmo tipo e não interfiram entre si, independente do caminho no grafo de fluxo de controle do programa.

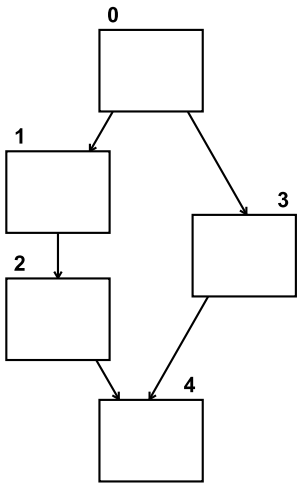

Figura 5: Exemplo de escolha dos pares de domínios ativos a serem unidos de acordo com grafo de fluxo de controle do programa

A seguir, calcula-se o custo da união para cada par de domínios ativos escolhido. Neste processo, primeiro é testado se a solução da função ϕ depende da solução de uma outra função ϕ, relativa ao mesmo domínio ativo, porém pertencente a outro bloco básico. A solução de uma função ϕ depende da solução de outra função ϕ, se em seu conjunto in[r], existir um elemento relativo ao domínio ativo com um estado de consistência X, que depende da solução da função ϕ do bloco anterior. Se a função não depender da solução de outra função ϕ, é feita uma análise de casos, já descrita na Seção 3.1. Esta análise de casos considera as referências que alcançam ϕ, ou seja, as contidas em *in[ϕ]*, e o fato da variável pertencente ao item in[r] estar viva ou não a partir deste ponto no programa. Todos os casos com as possíveis soluções de ϕ são testados e o de menor custo, menor número de instruções inseridas é a solução da função ϕ escolhida. Caso a função ϕ dependa da solução de uma outra função ϕ, é aplicada a heurística de força bruta para sua resolução.

O custo da união escolhida é inicializado com infinito. Todos os pares testados são avaliados, seus custos de união são comparados, e o que tiver um menor custo é então unido. Existe uma lista encadeada de estruturas que contêm os candidatos à união, assim como seu respectivo custo. Após a união, as instruções que dependem da solução de ϕ são então emitidas. A seguir, a iteração principal é feita novamente, porém a solução da função ϕ é propagada na análise de fluxo de dados RCR.

Finalmente, quando o número de domínios ativos existentes no programa for menor ou igual ao número de registradores de máquina, a iteração principal do programa pára e as variáveis pertencentes a todos os domínios ativos são substituídas pelos registradores atribuídos àquele domínio ativo. A escolha de qual registrador é alocado ao domínio ativo depende de vários

fatores: o tipo do domínio ativo, se ele deve ser alocado a um registrador *callee-saved* ou não, e os domínios ativos com os quais ele interfere. O registrador escolhido para o domínio ativo não deve ser igual ao registrador escolhido para o domínio ativo que interfere com ele, pois pelo conceito de interferência, dois domínios ativos que interferem entre si não podem ocupar o mesmo registrador.

A heurística LRO [12] usada para determinar de forma ótima e eficiente a solução para todas as funções ϕ no caso do grafo DG_ϕ ser acíclico, não foi explorada nesta versão da implementação. Isto porque, apesar de ela ser mais eficiente, ela só é usada em situações onde a solução de uma função ϕ depende de outra função ϕ e quando o grafo DG_ϕ é acíclico, o que não cobre todos os casos possíveis. Portanto, foi implementada apenas a heurística por força bruta, gulosa.

Primeiramente implementou-se um algoritmo que utiliza a última análise de casos descrita na Seção 3.1 para analisar as referências alcançadas por ϕ. A fim de se economizar uma passada pelo código do programa, procura-se as referências nas instruções sucessoras da função ϕ que possuam um registrador com uma variável em estado indefinido (X) devido a solução de ϕ ou que possuam (ω_i, σ_i) em seu conjunto *in*.

Esta implementação do algoritmo não gerou resultados eficientes quando os domínios ativos unidos estavam vivos simultaneamente e suas referências eram intercaladas dentro de um bloco básico. Para resolver o problema, propôs-se inserir interferências entre as variáveis vivas, sendo que domínios ativos que interferem entre si não podem ser unidos.

Também foi implementado a solução proposta por Ottoni e Araújo para os itens A1 e A2 da Seção 3.1, que faz com que a análise do custo da resolução das funções ϕ se estenda a todos os blocos básicos do programa, e não apenas considere as instruções sucessoras à função ϕ. Desta maneira, o custo da intercalação é computado também como o custo da solução de ϕ.

Os resultados destas implementações são discutidos na Seção 5.

4.4 Algoritmo de Alocação de Registradores Baseada em Crescimento de Domínio Ativo e Combinação de Registradores

O algoritmo do Método de Alocação de Registradores Baseado no Crescimento de Domínios Ativos após a inserção da transformação de Combinação de Registradores, discutida na Seção 5.2, e da extensão da análise do custo de união de dois domínios ativos proposta em [12] tem a seguinte forma:

```
make_webs;
live_ranges = webs;
make_coalesce;
make_webs;
live_ranges = webs;
insert_phis;
make_interferences;
create_DGgraph;
while (numLiveRanges > numRegs) do
        make_rcr;
        emit_instructions_independents_phi;
        solution_phi_function;
substitute_liveRanges_regs;

procedure solution_phi_function()
        form_pairs;
        for each pair do
                if not(phi depends on another phi) then
                        case_analysis;
                        trace_basic_blocks
                else
                        brute_force_heuristic;
                        case_analysis;
                        trace_basic_blocks
        cost = infinity;
        for each pair do
                if (pair->cost < cost) do
                        cost = pair->cost;
                        merge = cost;
        merge_pair(merge);
```

onde as funções inseridas no algoritmo são:

- `make_coalesce:`

  função que faz a transformação de Combinação de Registradores. Ela percorre o código do programa, procurando por instruções de *move*. Quando as encontra, se os domínios ativos fonte e destino da instrução de cópia não interferirem entre si, e o domínio ativo destino da cópia não for armazenado na memória em nenhuma instrução após a instrução de cópia, a instrução de cópia é removida e o operando usado pela instrução de cópia é substituído pelo operando destino da cópia em todo o programa. Ou seja, o domínio ativo que era usado na instrução de cópia é removido do programa, e suas referências substituídas por referências ao domínio ativo destino da instrução de cópia;

- `trace_basic_blocks:`

  função que acrescenta ao custo da união de dois domínios ativos o número de instruções que não dependem da solução de ϕ, logo não foram computados no custo original. Para tal, ela percorre todo o código do programa fazendo a análise de casos existentes na etapa

de emissão de instruções não dependentes da solução de funções ϕ, considerando que os domínios ativos tivessem sido unidos. Sempre que encontrar a inserção de uma instrução de acesso à memória, o custo da união dos domínios ativos em questão é incrementado.

4.4.1 Implementação do Algoritmo de Alocação de Registradores Baseada em Crescimento de Domínio Ativo e Combinação de Registradores

Um fato interessante pôde ser observado na implementação do algoritmo de Ottoni e Araújo [12]: o código resultante possuía muitas instruções de cópia desnecessárias. Analisando este resultado, percebe-se que ele acontece pela falta da aplicação da transformação de Combinação de Registradores *(Register Coalesce)*. Esta transformação é uma variação da transformação de propagação de cópia, muito utilizada na otimização de compiladores, e que elimina cópias de um registrador para outro. A transformação de Combinação de Registradores faz justamente a eliminação de instruções de cópia desnecessárias, substituindo a variável fonte da cópia pelo alvo da cópia, eliminando com isso a instrução de cópia e uma variável. O algoritmo de Ottoni e Araújo não implementa esta transformação.

Decidiu-se então, implementar a transformação de Combinação de Registradors no algoritmo, a fim de melhorar o código gerado por ele. A transformação foi implementada no início do método, após o cálculo das *webs* e dos domínios ativos. Após a aplicação da transformação, as *webs* e os domínios ativos são recalculados.

4.5 Ambiente da Implementação

Primeiramente, o compilador GCC [1] foi estudado para se implementar o Método Baseado em Crescimento de Domínios Ativos [12]. Resolveu-se por não utilizá-lo devido à sua complexidade.

Para testar o método de Alocação de Registradores Baseado em Crescimento de Domínios Ativos [12], ele foi implementado no Machine SUIF [16] (MachSUIF), uma estrutura flexível e extensível para a construção de *back-ends* de compiladores. Seus objetivos são: construção de novos passos de otimizações parametrizados em relação à máquina alvo e portáveis a outros ambientes de compilação; suporte para uma nova arquitetura alvo da compilação; suporte para o desenvolvimento de uma nova representação intermediária (linguagem intermediária) sem ter que reescrever todas as otimizações já desenvolvidas. MachSUIF é fácil de usar, é modular, facilita a introdução ou remoção de novos passos de otimização ou análise e permite o reúso de código para otimização. No MachSUIF, um *back-end* mínimo é composto pelos seguintes passos de compilação: geração de código de baixo nível, geração de código específico de máquina, alocação de registradores, término da geração de código específico de máquina, geração de código de montagem *(assembly)*. Vários passos de otimização podem ser inseridos durante essa seqüência, como escalonamento, desdobramento de constantes, passagem do código para a forma SSA, entre outros. O MachSUIF permite o desenvolvimento de novos passos de otimização e a introdução de novas arquiteturas de forma fácil. Ele foi desenvolvido adotando-se a filosofia de que os passos de otimização e análise fossem desenvolvidos de forma independente do ambiente de compilação e do alvo da compilação. Esta filosofia faz com que o MachSUIF e suas otimizações e análises sejam fácil e rapidamente inseridas em um novo ambiente de compilação.

O método de Alocação de Registradores utilizado pelo MachSUIF é o método de George e Appel [14], uma melhoria do algoritmo de Chaitin [7], também melhorando o desempenho do método de Briggs [4]. Este método intercala a fase de simplificação com a heurística de *coalesce* de Briggs, fazendo com que o algoritmo de alocação fique mais agressivo. Este método é denominado *Iterated Register Coalescing*. Este passo de compilação de Alocação de Registradores implementado no MachSUIF foi substituído pelo método implementado nesta dissertação, baseado em Crescimento de Domínios Ativos. Os códigos resultantes dos dois métodos de alocação foram comparados e o resultado é apresentado na Seção 6.

O processador alvo para os experimentos foi o Alpha, que tem como base uma arquitetura load/store do tipo RISC, com 64 registradores de máquina, dos quais 54 são alocáveis. Destes, 23 são registradores de propósito geral e 31 são de ponto flutuante. O critério de comparação entre os dois métodos de Alocação de Registradores, baseado em Crescimento de Domínios Ativos e baseado na técnica de George e Appel, foi o número de instruções de acesso à memória inseridas no código resultante da alocação, ou seja, número de instruções de *load* e *store*.

O *benchmark* utilizado para os testes comparativos foi constituído de programas que implementam algoritmos clássicos, como: quicksort, bubsort, multiplicação de matrizes, série de fibonacci, crivo de eratóstenes, busca por largura em grafos, entre outros. O *benchmark* foi escolhido considerando-se as estruturas, o grafo de fluxo de controle dos programas, o número de variáveis utilizadas pelo programa, tipo das variáveis. Isto faz com que vários tipos de grafos de fluxo de controle de programas sejam cobertos, o que afeta o número de funções ϕ a serem inseridas no código, e também a dependência entre funções ϕ. Logicamente, casos de não dependência e de dependência entre as funções ϕ foram testados.

4.6 Conclusão

Nesta seção foi descrito o trabalho prático contido nesta dissertação: a implementação, juntamente com as decisões de implementação, do Algoritmo Global de Alocação de Registradores Baseada em Crescimento de Domínios Ativos proposto por Ottoni e Araújo [12]. O algoritmo de Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos e Combinação de Registradores também descrito nesta seção, constitui uma contribuição ao método de Ottoni e Araújo. Os resultados das implementações são discutidos a seguir.

Capítulo 5

Avaliação dos Resultados

5.1 Introdução

Esta seção apresenta os resultados das implementações descritas na Seção 4. A Subseção 5.2 apresenta uma análise dos algoritmos implementados a partir dos resultados obtidos. A Subseção 5.3 mostra a execução passo a passo do trabalho desta dissertação, a partir da execução de um programa do *benchmark*. Finalmente, a Subseção 5.4 apresenta os resultados comparativos do método discutido nesta dissertação com o método de George e Appel [14], uma versão melhorada da abordagem de coloração de grafo de interferência proposta por Chaitin.

5.2 Análise dos Algoritmos

Como dito na Seção 4.3.1, para a escolha dos pares de domínios ativos a terem sua união testada, foi dada a preferência para a escolha de um domínio ativo de cada caminho no grafo do fluxo do programa que chega ao bloco básico que contém a função ϕ. Porém, em um dos programas presente no *benchmark*, o programa *integer* mostrado a seguir, esta escolha não gerou um resultado eficiente. O grafo de fluxo de controle deste programa é mostrado na Figura 6.

```
#define COUNT 10
main () {
        int j, k;
        long i;
        i = 0;
        while (i < COUNT) {
                j = 240; k = 15;
                j = (  k * ( j / k)  );
                j = (  k * ( j / k)  );
                j = (  k+k+k+k+ k+k+k+k+ k+k+k+k+ k+k+k+k  );
                k = (  j -k-k-k-k -k-k-k-k -k-k-k-k -k-k-k );
                j = (  k << 4 );   k = (  k << 4  );
                j = (  k * ( j / k )  );
                j = (  k * ( j / k )  );
                j = (  k+k+k+k+ k+k+k+k+ k+k+k+k+ k+k+k+k  );
                k = (  j -k-k-k-k -k-k-k-k -k-k-k-k -k-k-k );
                j = (  k << 4  ); k = (  k << 4  );
                j = (  k * ( j / k )  );
                j = (  k * ( j / k )  );
                j = (  k+k+k+k+ k+k+k+k+ k+k+k+k+ k+k+k+k  );
                k = (  j -k-k-k-k -k-k-k-k -k-k-k-k -k-k-k );
                i++;  }
        }
```

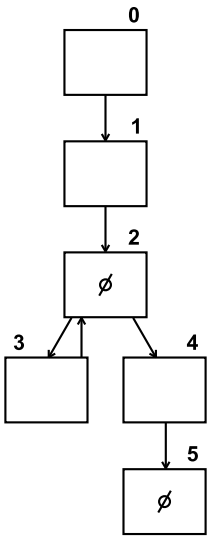

Figura 6: Grafo do fluxo de controle de integer.c

Pode-se observar no grafo do fluxo de controle, que o bloco que contém uma função ϕ é o Bloco Básico 2, que possui como antecessores os Blocos Básicos 1 e 3. Desta forma, escolheu-se um domínio ativo do Bloco Básico 1 e outro domínio ativo do Bloco Básico 3 para terem sua união testada. Observa-se que o trecho do programa *integer* referente ao Bloco Básico 1, mostrado

abaixo e já transformada na linguagem intermediária do MachSUIF para a arquitetura Alpha, tem apenas duas referências a um domínio ativo, no caso o *i* e o *vr0*:

```
main:
        .loc    2 15
        ldil    $vr0,0
        mov     $vr0,main.i
```

Após a transformação de Combinação de Registradores, Seção 4.4, introduzida no algoritmo original, o código do Bloco Básico 1 se torna:

```
main:
        .loc    2 15
        ldil    main.i,0
```

Como pode ser observado, o Bloco Básico 1 agora só possui uma referência a um domínio ativo, o *i*. Desta forma, este domínio ativo é unido com todos os outros cujas referências estão presentes no Bloco Básico 3. Neste caso são praticamente todos os outros domínios ativos presentes no programa, ou seja, *i* sempre será um dos domínios ativos a ter sua união testada. O problema surge porque o domínio ativo *i* está vivo ao longo do programa. Ele é usado no final do Bloco Básico 3. Desta forma, ao uní-lo com qualquer outro domínio ativo, 2 instruções de acesso à memória são necessárias: (a) um *store i* quando o outro domínio ativo que foi unido a ele for utilizado, e (b) um *load i* no final do Bloco Básico 3, quando ele for utilizado novamente. Esta situação ocorre na primeira e na terceira implementação do método, não ocorrendo na segunda implementação porque como *i* está vivo ao longo do programa, ele interfere com todas as outras variáveis do Bloco Básico 3, que estão vivas simultaneamente a *i*.

Ao executar o algoritmo de alocação de registradores baseado em crescimento de domínios ativos novamente, porém escolhendo-se todos os domínios ativos do mesmo tipo e que não interferem entre si para serem testados, independentemente do caminho no grafo de fluxo de controle do programa, este fato não ocorreu mais. Nenhuma instrução de acesso à memória foi inserida no código, uma vez que não foi escolhida a união de *i* com qualquer outro domínio ativo. Isto se deu devido ao seu custo de 2 instruções, enquanto várias outras uniões possuíam custo 0. Contudo, o problema com esta nova execução do algoritmo é que foram identificados 102 domínios ativos no programa *integer* após a aplicação da transformação de combinação de registradores, existindo assim mais de mil possibilidades de pares de domínios ativos, onde cada um deles deveria ser testado para cada iteração do programa, no caso 49, considerando a existência de 54 registradores de máquina. Chegou-se a um impasse: ou o alocador executa rapidamente, com um custo menor, e gera um código resultante menos eficiente com duas intruções de acesso à memória, ou então o alocador usa um algoritmo com um custo grande de recursos, porém gerando um código mais eficiente, sem nenhuma instrução de acesso à memória. A primeira opção é a mais interessante, assim para satisfazê-la, a escolha dos pares de domínios ativos a terem sua união testada continua sendo a união de domínios ativos presentes em caminhos diferentes que chegam ao bloco básico que contém a função ϕ.

O segundo resultado importante observado foi o fato de que o código resultante da Alocação de Registradores Baseada em Crescimento de Domínios Ativos possui um grande número de instruções de cópias desnecessárias, inclusive cópia de um registrador para si mesmo. Este grande número de instruções de cópia de registradores virtuais é causada por interferência do próprio compilador. Além disso, o número de domínios ativos encontrados era muito grande, bem maior que o número de candidatos à alocação encontrado pelo método de George e Appel [14]. Isto se deve ao fato de o algoritmo de Alocação Baseado em Crescimento de Domínios Ativos não implementar a transformação conhecida como Combinação de Registradores, o *Register Coalesce*. Esta tranformação é uma variação da propagação de cópia, que elimina cópias de um registrador para outro. Nesta tranformação, procura-se instruções de cópias de registradores no código intermediário, da forma $s_j \leftarrow s_i$, tal que tanto s_i como s_j não interferem um com o outro e nem s_j nem s_i são armazenados na memória entre a atribuição de cópia e o fim da rotina. Depois de se encontrar uma instrução deste tipo, a transformação de Combinação de Registradores procura a instrução que escreveu em s_i e a modifica para colocar seu resultado em s_j ao invés de s_i, e remove a instrução de cópia [18]. Desta maneira, s_i não existe mais no código do programa. Esta tranformação diminui o número de candidatos à alocação e remove instruções de cópia desnecessárias.

A Alocação de Registradores de Chaitin [7] implementa esta transformação, porém, a Alocação de Registradores Baseada no Crescimento de Domínios Ativos não. Decidiu-se então implementar a tranformação de Combinação de Registradores no método implementado, a fim de melhorar o código gerado, aumentando sua eficiência. A transformação foi implementada no início do método, depois de se calcular as *webs* e os domínios ativos (necessárias para se verificar a existência de interferências entre os possíveis candidatos à combinação). Após a transformação, as *webs* e os domínios ativos são recalculados. O resultado foi bastante satisfatório, reduzindo o número de domínios ativos em média, em cerca de 40%. Pôde-se observar, desta forma, que esta transformação é muito importante para este método, não só pela eliminação de instruções de cópia desnecessárias mas também pela diminuição do número de domínios ativos, o que faz com que a alocação se torne mais rápida.

Para ilustrar, veja por exemplo a função *qsort*, mostrada a seguir na linguagem C, que corresponde ao algoritmo *quicksort*:

```
void qsort( i, j)
    int i,j;
{   int pivot;
    int pindex;
    int k;
    int first;
    int l;
    int r;
    int t;

    /* find pivot */
    pindex = 0;
    first = a[i];
    k = i+1;
    while ((pindex == 0) && (k <= j))
    {   if (a[k] > first)
        {
            pindex = k;
        }
        else if (a[k] < first)
        {
            pindex = i;
        }
        k = k + 1;
    }
    if (pindex != 0)
    {   pivot = a[pindex];
        /* partition */
        l = i;
        r = j;
        do
        {   /* swap */
            t = a[l];
            a[l] = a[r];
            a[r] = t;
            while (a[l] < pivot)
                l = l + 1;
            while (a[r] >= pivot)
                r = r - 1;
        }
        while (l <= r);
        k = l;
        qsort( i, k-1);
        qsort( k, j);
    }
}
```

A função *qsort* passada para a linguagem intermediária do MachSUIF para a arquitetura Alpha antes do passo de alocação de registradores, é mostrada no item A.1 do Apêndice.

Antes de implementar a transformação de Combinação de Registradores no Método de Alocação de Registradores Baseado em Crescimento de Domínios Ativos, o método identificou 109 domínios ativos. Além disto, existem várias instruções de cópia desnecessárias no código, instruções de *move*, como por exemplo a quarta instrução do código:

```
mov     $vr0,qsort.pindex
```

O código contém 156 instruções. Após a implementação da Combinação de Registradores, o número de domínios ativos encontrados caiu para 64 *webs*, além disso, as instruções de *move* desnecessárias foram removidas do código, o código resultante contém 112 instruções, ou seja, 44 instruções de cópia desnecessárias foram removidas.

O terceiro resultado importante observado foi o fato de que, em algumas situações, o código gerado pela primeira implementação do método baseado em Crescimento de Domínios Ativos apresentado na Seção 4.3, feita com o objetivo de se economizar uma passada pelo código do programa, continha instruções de *store* e *load*, enquanto que o gerado pelo método de coloração de grafo não continha nenhuma instrução de acesso à memória, ou seja, o método apresentava resultado pior que o de coloração de grafo.

Analisando-se estas situações, pôde-se observar que este fato ocorria sempre que um dos domínios ativos unidos na iteração estava presente nos dois caminhos do grafo de fluxo de controle do programa que chegava até as funções ϕ analisadas, e que as referências às variáveis pertencentes aos domínios ativos apareciam intercaladas no bloco básico que possuía referências a ambos, ou seja, estavam vivas simultaneamente em certa parte do bloco básico que as continha. Além disso, uma das referências permanecia viva ao final deste bloco básico. Ao se calcular o custo da união dos domínios ativos, esse custo era mínimo, pois é o custo da solução da função ϕ relativa a esta união. A solução da função ϕ relativa a esta união não retratava o custo das instruções de *store* e *load* inseridas porque para a solução de uma função ϕ na primeira implementação do método, são analisadas e consideradas as referências que alcançam e que são alcançadas pela função ϕ, e isto representa as últimas referências do bloco básico antecessor ao bloco que contém a função ou então a primeira instrução do próprio bloco ou seus sucessores que usam ou definem os domínios ativos em questão. O fato é que, neste caso, o bloco básico antecessor, o sucessor, ou o próprio bloco básico, continha os dois domínios ativos intercalados, porém apenas a última referência a um destes domínios ativos é considerada para a solução da função ϕ, no caso do bloco antecessor, e a primeira referência a um destes domínios ativos é considerada para a solução de ϕ no caso do bloco sucessor ou do próprio bloco. Desta forma, o custo da troca de variáveis do domínio ativo a ocupar o registrador durante o bloco básico antecessor não é analisada. Resumindo, esta situação ocorria quando as variáveis pertencentes aos domínios ativos unidos estavam vivas ao mesmo tempo.

Um exemplo desta situação pode ser observado no código mostrado pela Figura 7. Considere a união sendo testada como a dos domínios ativos que contêm x e w. Para se resolver a função ϕ existente no Bloco Básico 14, considere as referências que a alcançam e as que são alcançadas por ela. As referências, dos domínios ativos em questão, que alcançam ϕ são uma definição de x no Bloco Básico 13 e um uso de x no Bloco Básico 15 (blocos antecessores de 14). Enquanto a

referência dos domínios ativos em questão que é alcançada por ϕ é um uso de x no Bloco Básico 14. Logo, analisando o custo das possíveis soluções de ϕ, vê-se que a de menor custo é o par *(x, I)*, ou seja, a própria variável x em estado de consistência I, gerado pela sua definição, que tem custo 0 (zero). Porém, na próxima iteração do algoritmo, após x e w serem unidos, na etapa de emissão de instruções que não dependem da solução de nenhuma função ϕ, tem-se que, ao final do Bloco Básico 14, o registrador alocado ao domínio ativo unido em questão (contendo x *e* w), contém a variável x em um estado inconsistente com a memória. A próxima referência a uma das variáveis do domínio ativo é uma definição de w no início do Bloco Básico 15. Como o registrador contém x em um estado inconsistente, e x está viva, tem-se que inserir uma instrução de *store* x antes da instrução de definição de w. Após a definição de w há um uso do mesmo, e após este uso, há um uso de x. O registrador agora contém w em um estado inconsistente com a memória, como w não está mais vivo, tem-se que inserir uma instrução de *load* x, antes de seu uso, ou seja, tem-se que carregar o valor de x da memória para que ele seja usado. Assim, a união de x e w teve um custo de 2 instruções de acesso à memória inseridas que não foi captado na resolução da função ϕ, só captado na iteração seguinte, na fase de emissão de instruções não dependentes de funções ϕ.

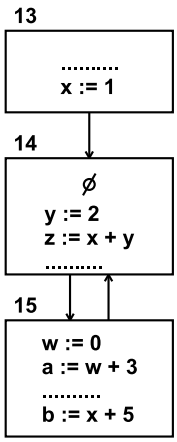

Figura 7: Exemplo de programa que gera código ineficiente no Método Baseado em Crescimento de Domínios Ativos

Para se resolver esta deficiência do algoritmo implementado, tentou-se a seguinte solução: foram inseridas interferências entre variáveis vivas, e domínios ativos que interferem um com o outro não podem ser unidos. Por exemplo, na Figura 7 mostrada, como x *e* w estão vivos ao mesmo tempo, é inserida uma interferência entre os domínios ativos que as contêm, logo, estes domínios ativos não poderão ser unidos. Esta resolução eliminou este problema, sendo que o código gerado pelo Método de Alocação Baseado em Crescimento de Domínios Ativos se tornou eficiente também nestas situações. Esta resolução do problema combina a solução dada pelo Método de Coloração de Grafos, que trabalha com interferência entre variáveis vivas com a solução dada pela Combinação de Domínios Ativos.

A seguir, foi implementado o algoritmo completo, como proposto por Ottoni e Araújo [12], onde a análise do custo da resolução das funções ϕ se estende a todos os blocos básicos do programa, e não apenas considerando as referências que alcançam ou são alcançadas por ϕ. Ou seja, a etapa de emissão de instruções que não dependem da resolução de uma função ϕ é considerada, considerando que os domínios ativos tendo a união testada já estivessem unidos. O custo da união de dois domínios ativos é feito normalmente, considerando a solução da função ϕ, porém após o cálculo deste custo, mais uma etapa é inserida no algoritmo. Esta etapa percorre todos os blocos básicos do programa, somando ao custo da união as intruções inseridas no passo de emissão de instruções não dependentes da solução de funções ϕ (instruções são inseridas fazendo a análise de casos já citada na explicação do algoritmo), considerando que os dois domínios ativos fossem unidos. Se alguma instrução for inserida, o número destas instruções inseridas é somado ao custo da união dos domínios ativos. Desta forma, o custo da intercalação das variáveis é computado, o custo da união aumenta, e possivelmente esta união não é escolhida. No exemplo da Figura 7, o custo da união de x e w é 2, pois a etapa de emissão de instruções não dependentes de funções ϕ se x e w são unidos é considerada, logo o custo da inserção do *store* x e *load* x no Bloco Básico 15 é considerado. Possivelmente existe uma outra união de menor custo, que então, é a escolhida. Esta solução, com certeza aumenta o custo do programa, mas resolve o problema explicado anteriormente, sem se aproximar da solução adotado pela coloração de grafos.

Ambas as soluções foram implementadas e testadas no Método de Alocação Baseado em Crescimento de Domínios Ativos, e o código gerado se tornou eficiente. Deu-se preferência a segunda solução, mais apropriada ao algoritmo, como proposta por Ottoni e Araújo [12], distanciando da solução adotada pelo Método de Coloração de Grafos.

O resultado final e mais importante foi o fato de que o Algoritmo de Alocação de Registradores Baseado em Crescimento de Domínios Ativos obteve resultados mais eficientes do que o Algoritmo de Coloração de Grafos em certas situações. Analisando-se estas situações, observou-se que este fato ocorre em dois casos distintos:

Caso 1: na presença de muitos laços aninhados. Neste caso, os limites dos laços foram derramados para a memória, enquanto vários registradores não foram usados dentro do laço. Uma situação semelhante é descrita por Briggs, Cooper e Torczon [5] no trabalho *Improvements to Graph Coloring Register Allocation*. No Caso 1, existem domínios ativos que representam variáveis globais do programa que permanecem vivas ao longo de todo o programa, sendo inclusive usadas dentro dos laços aninhados. Estes domínios ativos possuem um custo de derramamento mais alto devido as várias interferências existentes entre eles e outras variáveis nos grafos de interferência e por eles serem referenciados dentro dos laços. O alocador derrama então os domínios ativos com um custo menor, que são os limites dos laços.

O programa de multiplicação de matrizes, *matrixmul* ilustra esta situação. O seu algoritmo em C, no qual pode-se observar a existência de três laços aninhados em sequência, tem a seguinte forma:

```
    float a[100][100];
    float b[100][100];
    float c[100][100];
    float d[100][100];
void mm( n)
    int n;  {
    int i,j,k, l;
    float sum;
    float minus;
    int m;
    m = n;
    minus = 0;
    l = 0;
    while (l < n) {
       i = 0;
       while (i < n) {
          j = 0;
          while (j < n) {
              sum = 0;
              k = 0;
              while (k < n) {
                  minus = minus + (a[l][i]*b[k][j]);
                  sum = sum + (a[i][k]*b[k][j]);
                  k++;  }
              c[i][j] = sum;
              j++;  }  i++; }  l++;  }   }

main ()  {
    int n, i, j;
    read( &n);
    i = 0;
    while (i < n) {
        j = 0;
        while (j < n) {
            fread( &a[i][j]);
            j++;  }  i++;   }
    i = 0;
    while (i < n) {
        j = 0;
        while (j < n) {
            fread( &b[i][j]);
            j++;  } i++; }
    mm( n);
    printf("Result matrix:\n"); i = 0;
    while (i < n) {
        j = 0;
        while (j < n) {
            printf( "%f, ", c[i][j]);
```

```
    j++;  }  printf("\n");
i++;  }  }
```

O grafo de fluxo de controle da rotina *main*, na qual os limites dos laços foram derramados tem a forma mostrada na Figura 8:

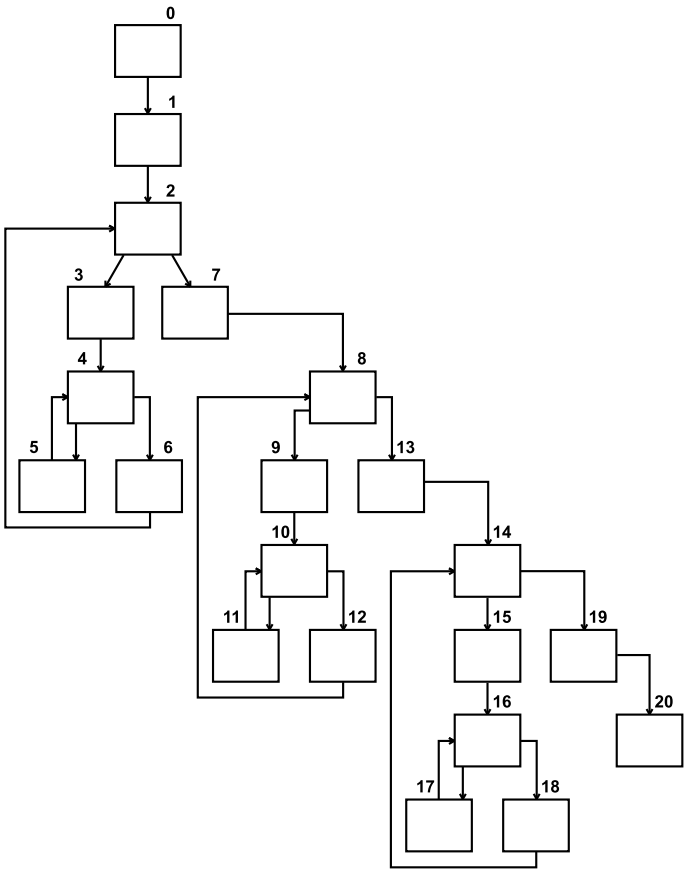

Figura 8: Grafo de fluxo de controle da função main do matrixmul.c

Neste programa, as matrizes *a*, *b*, *c* são variáveis globais vivas ao longo de todo o programa, sendo inclusive referenciadas dentro dos laços aninhados. O alocador baseado em coloração de grafos gerou 7 *loads* da variável *n* nos seguintes Blocos Básicos: 2, 4, 8, 10, 13, 14, 16. Pode-se observar que estes blocos pertencem a laços do programa.

Caso 2: quando variáveis globais permanecem vivas e muito usadas ao longo de todo o programa, sendo muito usadas dentro de laços. Neste caso o alocador baseado em crescimento de domínios ativos gera código mais eficiente do que o baseado em coloração de grafos. Nesta situação, os índices e as variáveis limite dos laços são derramadas para a memória por possuírem um custo menor de derramamento. Porém, o laço continua demandando registradores, o que acarreta o derramamento das variáveis globais que são muito referenciadas dentro dos laços, apesar

de estas possuírem um custo de derramamento maior. O exemplo do programa *whetsto*, mostrado a seguir, em linguagem C, ilustra este caso:

```
#define ITERATIONS      50 /* 5 Million Whetstone instructions */
double          x1, x2, x3, x4, x, y, z, t;
double          e1[4];
int             i, j, k, l, n4, n6, n10, n11;
main() {
        t   =   0.499975;
        n4  = 345 * ITERATIONS;
        n6  = 210 * ITERATIONS;
        n10 =   0 * ITERATIONS;
        n11 =  93 * ITERATIONS;
        j = 1;
        i = 1;
        while (i <= n4) {
                if (j == 1)  j = 2;
                else         j = 3;
                if (j > 2)   j = 0;
                else         j = 1;
                if (j < 1 )  j = 1;
                else         j = 0;
                i++;  }
        j = 1;
        k = 2;
        l = 3;
        i = 1;
        while (i <= n6) {
                j = j * (k - j) * (l -k);
                k = l * k - (l - j) * k;
                l = (l - k) * (k + j);
                e1[l - 2] = j + k + l;          /* C arrays are zero based */
                e1[k - 2] = j * k * l;
                i++;  }
        j = 2;
        k = 3;
        i = 1;
        while (i <= n10) {
                j = j + k;
                k = j + k;
                j = k - j;
                k = k - j - j;
                i++;  }   }
```

O grafo de fluxo de controle do programa tem a forma mostrada na Figura 9.

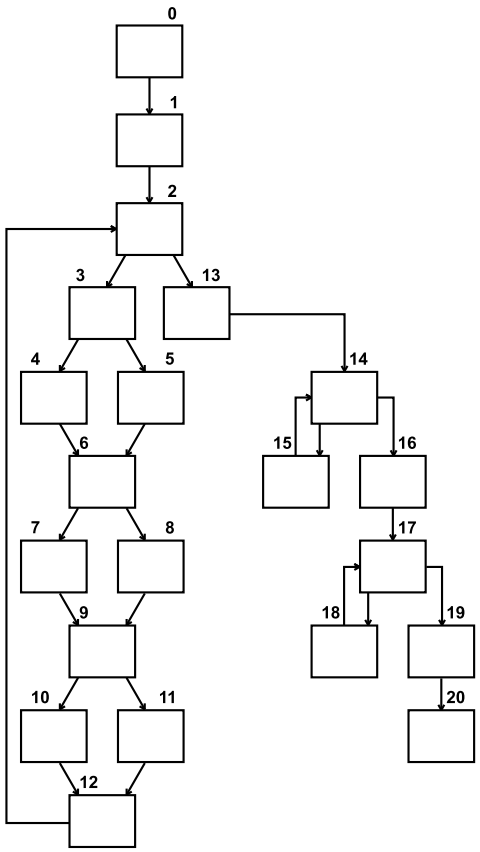

Figura 9: Grafo de fluxo de controle do whetsto1.c

O alocador baseado em coloração de grafos gerou 41 instruções de *load* e 29 instruções de *store*. Em todos os laços foram geradas instruções de derramamento do índice i e dos limites $n4$, $n6$ e $n10$ nos Blocos Básicos 2, 14 e 17, respectivamente. Os dois últimos laços do programa referenciam muito as variáveis globais k, j e l, principalmente o primeiro destes laços. E é justamente neste laço que concentra o maior número de derramamentos destas três variáveis, 23 instruções de *load*. O último laço obteve um acréscimo de 8 instruções de derramamento destas variáveis.

O alocador baseado em crescimento de domínios ativos não gera instruções de acesso a memória nestes casos pois é baseado em um princípio completamente distinto do de coloração de grafos, como já explicado.

5.3 Exemplo da Execução do Algoritmo de Alocação de Registradores Baseado em Crescimento de Domínios Ativos e Combinação de Registradores

Como um exemplo da evolução de nosso trabalho, são mostrados nesta seção todos os passos seguidos para um dos algoritmos do *benchmark* testado, o algoritmo de *bubsort*. O algoritmo na

linguagem C é mostrado a seguir:

```
#define UPPER 7;
main()
{   int top;
    int i;
    int temp;
    int n;
    int j;
    int list[7];
    j = 1;
    n = 7;
    i = 0;
    list[0] = 12;
    list[1] = 56;
    list[2] = 34;
    list[3] = 1;
    list[4] = 6;
    list[5] = -1;
    list[6] = 9;
    top = UPPER - 1;
    while (top > 1) {
        i = 1;
        while (i < top) {
            if (list[i] > list[i+1]) {
                /* integrate procedure swap */
                temp = list[i+1];
                list[i+1] = list[i];
                list[i] = temp;
            }
            i = i + 1;
        }
        top = top - 1;
    }
}
```

O grafo de fluxo de controle do programa *bubsort* tem a forma mostrada pela Figura 10.

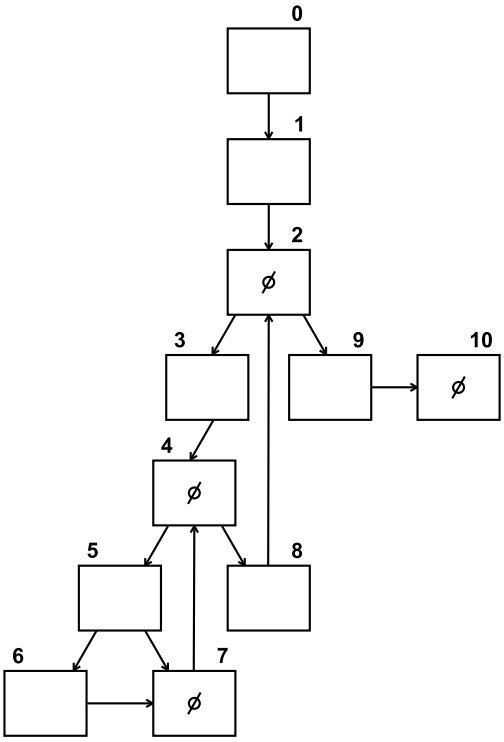

Figura 10: Grafo de fluxo de controle do bubsort.c

O algoritmo *bubsort* passado para a linguagem intermediária do MachSUIF para a arquitetura Alpha tem a forma mostrada no Item A.2 do Apêndice. Ressalta-se que os blocos básicos correspondentes ao trecho de código estão marcados.

A primeira implementação do algoritmo de Alocação de Registradores Baseado em Crescimento de Domínios Ativos identificou 109 domínios ativos. O primeiro passo do trabalho foi a inserção no algoritmo da tranformação de Combinação de Registradores (Seção 4.4). Após a implementação desta transformação, o algoritmo identificou 81 domínios ativos. Os domínios ativos unidos nas iterações são os mostrados na Tabela 1:

Iteração	Domínios Ativos Unidos
1	j e $vr90
2	n e top
3	$vr65 e $vr50
4	$vr64 e $vr52
5	$vr66 e $vr53
6	$vr68 e $vr49
7	$vr69 e $vr57
8	$vr63 e $vr56
9	temp e $vr58
10	$vr72 e $vr60
11	$vr74 e $vr61
12	$vr75 e $vr55
13	$vr76 e $vr102
14	i e $vr76 e $vr102
15	$vr86 e $vr75 e $vr55
16	$vr85 e $vr74 e $vr61
17	$vr83 e $vr72 e $vr60
18	$vr82 e $vr63 e $vr56
19	$vr81 e $vr69 e $vr57
20	$vr79 e $vr68 e $vr49
21	$vr77 e $vr66 e $vr53
22	$vr78 e $vr64 e $vr52
23	$vr71 e $vr65 e $vr50
24	$vr46 e j e $vr90
25	$vr44 e n e top
26	$vr43 e $vr46 e j e $vr90
27	$vr41 e $vr44 e n e top
28	$vr40 e $vr43 e $vr46 e j e $vr90

Tabela 1: domínios ativos unidos na primeira implementação do algoritmo

Esta foi a primeira implementação do Algoritmo de Alocação de Registradores Baseado em Crescimento de Domínios Ativos (veja Seção 4.3). Esta implementação gerou 4 instruções de acesso à memória na iteração 14. O domínio ativo *i* foi unido com o domínio ativo que continha *$vr76* e *$vr102*. Como *i* está vivo simultaneamente a estes dois domínios ativos e referências a eles são intercaladas, em certo momento no Bloco Básico 6, o registrador alocado a esta união continha *i*, como esta variável está viva, ela deve ser armazenada na memória para que o registrador seja alocado a *$vr102*. Depois de todas as referências para *$vr102* existe uma para *i*. Como *$vr102* não está mais vivo, tem-se que somente carregar o registrador com *i*. A seguir tem-se referências para *$vr76*. Neste caso, *i* deve então ser armazenado na memória, *$vr76* é alocado ao registrador, e após todas as suas referências, como ele não está mais vivo, *i* é carregado novamente ao registrador,

pois é usado a seguir. O código resultante desta implementação é mostrado no Item A.3 do Apêndice.

Na segunda implementação do algoritmo (veja Seção 4.3.1), foram inseridas interferências entre variáveis vivas simultaneamente, para eliminar as instruções de acesso à memória geradas pela primeira implementação. Os domínios ativos unidos nas iterações são os mostrados na Tabela 2:

Iteração	Domínios Ativos Unidos
1	j e $vr90
2	n e top
3	$vr65 e $vr50
4	$vr64 e $vr52
5	$vr66 e $vr53
6	$vr68 e $vr49
7	$vr69 e $vr57
8	$vr63 e $vr56
9	temp e $vr58
10	$vr72 e $vr60
11	$vr74 e $vr61
12	$vr75 e $vr55
13	$vr76 e $vr102
14	$vr86 e $vr76 e $vr102
15	$vr85 e $vr75 e $vr55
16	$vr83 e $vr74 e $vr61
17	$vr82 e $vr72 e $vr60
18	$vr81 e $vr63 e $vr56
19	$vr79 e $vr69 e $vr57
20	$vr77 e $vr68 e $vr49
21	$vr78 e $vr66 e $vr53
22	$vr71 e $vr64 e $vr52
23	$vr46 e j e $vr90
24	$vr44 e n e top
25	$vr43 e $vr46 e j e $vr90
26	$vr41 e $vr44 e n e top
27	$vr40 e $vr43 e $vr46 e j e $vr90
28	$vr38 e $vr41 e $vr44 e n e top

Tabela 2: domínios ativos unidos na segunda implementação do algoritmo

Esta implementação não gerou instruções de acesso à memória no código resultante. Pode-se notar que o domínio ativo i não foi unido com o que contém *$vr76* e *$vr102*, pois uma interferência foi inserida entre estes domínios ativos, fazendo com que eles não pudessem ser unidos. O código resultante deste método é mostrado no Item A.4 do Apêndice.

Na terceira implementação do algoritmo (veja Seção 4.4) todos os blocos básicos são pesquisados para se calcular o custo da solução de uma função ϕ. Os domínios ativos unidos nas iterações são os mostrados na Tabela 3:

Iteração	Domínios Ativos Unidos
1	j e $vr90
2	n e top
3	$vr65 e $vr50
4	$vr64 e $vr52
5	$vr66 e $vr53
6	$vr68 e $vr49
7	$vr69 e $vr57
8	$vr63 e $vr56
9	temp e $vr58
10	$vr72 e $vr60
11	$vr74 e $vr61
12	$vr75 e $vr55
13	$vr76 e $vr102
14	$vr86 e $vr76 e $vr102
15	$vr85 e $vr75 e $vr55
16	$vr83 e $vr74 e $vr61
17	$vr82 e $vr72 e $vr60
18	$vr81 e $vr63 e $vr56
19	$vr79 e $vr69 e $vr57
20	$vr77 e $vr68 e $vr49
21	$vr78 e $vr66 e $vr53
22	$vr71 e $vr64 e $vr52
23	$vr46 e j e $vr90
24	$vr44 e n e top
25	$vr43 e $vr46 e j e $vr90
26	$vr41 e $vr44 e n e top
27	$vr40 e $vr43 e $vr46 e j e $vr90
28	$vr38 e $vr41 e $vr44 e n e top

Tabela 3: domínios ativos unidos na terceira implementação do algoritmo

Pode-se observar que os domínios ativos unidos foram os mesmos que os unidos pelo algoritmo com interferências entre variáveis vivas simultaneamente. Neste caso, a união do domínio ativo i com o que contém *$vr76* e *$vr102* teve um custo de solução da função ϕ presente no Bloco Básico 7 igual a 4, o número de instruções de acesso à memória inseridas na primeira implementação. Desta forma, como existiam uniões de menor custo, elas foram as escolhidas. O código resultante desta implementação é o mostrado no Item A.5 do Apêndice.

Pode-se observar que os resultados da segunda e da terceira implementação foram semelhantes.

5.4 Resultados Comparativos

A Tabela 4, mostra para cada um dos algoritmos no *benchmark* testado, o número de instruções de *load* e *store* obtido, na ordem das colunas: (I) pelo método de coloração de grafos, (II) primeira implementação do método baseado crescimento de domínios ativos sem considerar todos os blocos básicos no custo de ϕ, (III) pela implementação do método baseado em crescimento de domínios ativos com interferência entre variáveis vivas, (IV) pela implementação do método que considera todos os blocos básicos no custo de ϕ [12]. Na Tabela 4, cada uma dessas implementações são representadas por CG, 1 CDA, 2 CDA e 3 CDA respectivamente.

Algoritmo Testado	CG	1 CDA	2 CDA	3 CDA
teste	0	0	0	0
fibonacci	0	0	0	0
bubsort	0	4	0	0
quicksort	2	0	0	0
bfirst	4	2	0	0
integer	0	2	0	2
knight	0	0	0	0
float1	0	0	0	0
matrixmul	7	0	0	0
matrixmul2	21	0	0	0
point2	0	0	0	0
rsieve	0	0	0	0
whetsto	70	4	0	0
whetsto3	139	-	6	6
gauss-seidel	3	0	0	0

Tabela 4: número de instruções de *load* e *store*

No programa *integer*, a primeira e a terceira implementação do método de alocação baseado em crescimento de domínios ativos geraram instruções de acesso à memória pelo motivo explicado no início desta seção, pois a escolha dos domínios ativos a serem unidos foi a escolha de domínios ativos de caminhos disjuntos. Já a segunda implementação, com interferência entre variáveis vivas, não gerou instruções de acesso à memória, uma vez que os pares a serem unidos foram formados de uma maneira diferente, pois o único domínio ativo a ter uma referência no Bloco Básico 1 da Figura 6 interfere com todos os outros domínios ativos do Bloco Básico 3. Logo, a escolha dos pares foi feita pela combinação de todos os domínios ativos existentes. Apesar de nenhuma instrução de acesso à memória ter sido gerada, a execução do método se tornou lenta e custosa.

A Tabela 5 mostra o número de domínios ativos identificados nos algoritmos presentes no *benchmark* testado antes da implementação da transformação de Combinação de Registradores *(Register Coalesce)* no algoritmo de Alocação Baseada em Crescimento de Domínios Ativos, e após a implementação da Combinação de Registradores no algoritmo. Observa-se que o número

dos domínios ativos mostrado na Tabela 5 para cada algoritmo testado é a soma dos domínios ativos de todas as suas rotinas.

Algoritmo Testado	Sem *Coalesce*	Com *Coalesce*
teste	21	12
fibonacci	19	8
bubsort	109	81
quicksort	109	64
bfirst	140	66
integer	142	102
knight	171	99
float1	41	20
matrixmul	149	83
point2	132	70
rsieve	55	27
whetsto	102	64
whetsto3	198	134

Tabela 5: número de domínios ativos antes e depois da transformação de *Coalesce*

Ressalta-se que medições de tempo de execução dos dois métodos não foram efetuadas devido ao fato de que o Método Baseado em Crescimento de Domínios Ativos não foi otimizado, seu estudo e análise está apenas começando. O Método de Coloração de Grafos vem sendo analisado e otimizado desde 1980, o que seria uma comparação injusta.

Capítulo 6

Conclusão

Neste trabalho, foi estudada uma solução alternativa para o problema de Alocação Global de Registradores. O problema de alocação de registradores é um dos problemas fundamentais de otimização de código. A solução mais empregada é a coloração do grafo de interferência [7]. Esta abordagem representa os conflitos entre os valores vivos, valores que ainda serão utilizados pelo programa, com um dado número de cores, que representam o número de registradores disponíveis na máquina. Os valores que não possuem cores são derramados para a memória. Este trabalho foi posteriormente estendido por várias outras contribuições, entre elas [4] e [14] . Apesar de este ser o método padrão adotado para a solução deste problema, vários pesquisadores ainda tentam encontrar métodos mais eficientes para a alocação de registradores. Isto se deve ao fato de que, em algumas ocasiões, a alocação de registradores por coloração de grafos produz trechos de código de baixa qualidade.

A solução alternativa estudada neste trabalho se chama Alocação Global de Registradores Baseada no Crescimento de Domínios Ativos. Ela foi proposta por Ottoni e Araújo em [12], e seu objetivo foi estudar o problema de alocação global de registradores de endereçamento para referências a vetores dentro de laços de programas para DSPs, implementando uma solução baseada em crescimento de domínios ativos. A técnica foi implementada com sucesso para este problema.

O Método para Alocação Global de Registradores Baseado em Crescimento de Domínios Ativos foi implementado por nós no MachSUIF de diversas formas. O código gerado por este método foi comparado ao código gerado pelo Método de Coloração de Grafos de George e Appel [14] quanto ao número de instruções que fazem acesso à memória no código resultante.

O primeiro resultado obtido por esta comparação mostrou que a transformação de Combinação de Registradores era necessária para otimizar o Método Baseado em Crescimento de Domínios Ativos. O código resultante do método continha muitas instruções de cópia desnecessárias. Também observou-se que o número de *webs* identificadas pelo método de coloração de grafos era menor que o número de *webs* identificados pelo método implementado. A transformação de Combinação de Registradores foi então implementada no algoritmo, e pôde-se observar uma redução de cerca 40% no número de domínios ativos identificados pelo método. Esta redução é de grande valia, pois quanto menor é o número de domínios ativos identificados pelo algoritmo,

mais rapidamente ele é executado. Também observou-se a eliminação de instruções de cópia desnecessárias.

A fim de reduzir o tempo de execução e a complexidade do algoritmo, diminuindo uma fase de leitura do código do programa, primeiramente foi implementado um algoritmo que, na análise das referências sendo alcançadas por ϕ para a solução desta função, procuram-se as referências nas instruções sucessoras da função ϕ que possuam um registrador com uma variável em estado indefinido (X) devido a solução de ϕ ou que possuam (ω_i, σ_i) em seu conjunto *in*. Foi observado que esta primeira implementação do método gerava código de pior qualidade em certas situações. Estas situações eram freqüentes a vários programas testados, analisando-as descobriu-se que o método falhava na captura do custo de união de dois domínios ativos quando ambos estavam presentes intercalados no mesmo bloco básico, sendo que pelo menos um deles permanecia vivo no final do bloco básico. Logo, o custo da união calculado era mínimo, pois a solução da ϕ resultava na referência que alcançava e era alcançada por ϕ. Este cálculo não considerava o custo gerado pela intercalação das referências no mesmo bloco básico. Apenas na próxima iteração do método, depois que os domínios ativos tinham sido unidos com um custo mínimo, este custo da intercalação era "descoberto", ou seja, as instruções que não dependiam da solução de ϕ eram emitidas.

A segunda implementação do método foi feita conforme a proposta por Ottoni e Araújo [12], onde o cálculo do custo da união de dois domínios ativos não considera apenas as referências aos domínios ativos que alcançam ou são alcançadas pela função ϕ, mas sim se estende a todo o programa. Logo, após o custo da solução da função ϕ ser calculado, todo o código do programa é percorrido, analisando se alguma instrução seria emitida caso os domínios ativos em questão fossem realmente unidos. Para toda instrução inserida descoberta neste passo, o custo da união é incrementado. Desta maneira, o custo da união de dois domínios ativos reflete o custo real da união, incluindo o custo das instruções que não dependem da solução de ϕ. O código gerado por esta segunda implementação do método se tornou tão eficiente quanto o gerado pelo Método de Coloração de Grafos.

Uma solução para o problema gerado na primeira implementação do Método Baseado em Crescimento de Domínios Ativos também foi proposta e implementada: inserção de interferências entre domínios ativos vivos simultaneamente, sendo que domínios ativos que interferem entre si não podem ser unidos. É uma solução eficiente, que combina a solução adotada pela Coloração de Grafos com a solução adotada pelo Crescimento de Domínios Ativos. Esta implementação foi a mais eficiente entre as três: gera código de melhor qualidade com um tempo de execução menor que o algoritmo completo de Ottoni e Araújo.

O resultado mais importante obtido por este trabalho, porém, foi o fato do Algoritmo Baseado em Crescimento de Domínios Ativos ser mais eficiente do que o Algoritmo Baseado em Coloração de Grafos em relação ao critério de número de instruções de acesso à memória geradas em situações nas quais existem muitos laços aninhados e muitas variáveis globais que permanecem vivas durante todo o programa e são referenciadas dentro destes laços, sendo eles aninhados ou não. Enquanto o alocador baseado em coloração de grafos gera muitas instruções de derramamento, o alocador baseado em crescimento de domínios ativos não gera instruções de acesso à memória nestes casos. Isto se deve ao fato de as decisões de derramamento da coloração de grafos serem estimadas, e não se sabe ao certo o que acontece após estes derramamentos, mesmo porque não existe uma análise

de fluxo de controle neste método. Já o método baseado em crescimento de domínios ativos, além de fazer análise de fluxo de controle, sabe exatamente o que ocorre se dois domínios ativos são unidos, logo suas decisões de derramamento são mais reais, baseadas em medições reais.

Observou-se também que o Método de Coloração de Grafos de George e Appel é computacionalmente mais eficiente que o Método Baseado em Crescimento de Domínios Ativos em relação ao tempo de execução: este último tem que percorrer o código do programa pelo menos 3 vezes a cada iteração. Uma para fazer a análise de consistência e alcançabilidade de registradores, outra para a emissão de instruções não dependentes de funções ϕ, e a terceira para a resolução das funções ϕ para cada par de domínios ativos testado. Além disso, o método também percorre o código do programa para construir o grafo de dependência entre funções ϕ (DG_ϕ). Isto aumenta o tempo gasto na execução do método, aumentando sua complexidade.

Também conclui-se que o Método Baseado em Crescimento de Domínios Ativos é mais eficiente quanto ao número de instruções de *load* e *store* geradas que o Método de George e Appel nos casos citados. Existe, portanto, um compromisso qualidade de código X custo a ser considerado para a escolha entre o método de Coloração de Grafos (mais barato) e o método baseado em Crescimento de Domínios Ativos (melhor qualidade).

Especificamente, as contribuições desta dissertação de mestrado são:

1. Implementação de três versões do algoritmo desenvolvido por Araújo e Ottoni para Alocação Global de Registradores Baseada em Crescimento de Domínios Ativos:
   (a) a primeira faz uma tentativa de se economizar uma passada pelo algoritmo, mas não se mostrou eficiente no cálculo do custo da união de dois domínios ativos quando estes apareciam intercalados em um mesmo bloco básico;
   (b) a segunda insere interferências entre domínios ativos vivos simultaneamente para evitar os problemas encontrados na primeira implementação. Gerou código eficiente, porém se aproxima do método de Coloração de Grafos;
   (c) a terceira foi a implementação na íntegra do método proposto por Ottoni e Araújo, que dá uma passada a mais pelo código do programa para calcular o custo da união dos domínios ativos. Gerou código tão eficiente quanto o método de Coloração de Grafos, porém se mostrou computacionalmente mais cara.
2. Modificação do algoritmo de Ottoni e Araújo para incorporar a transformação de Combinação de Registradores;
3. Comparação dos resultados das três implementações do método de Alocação Baseada em Crescimento de Domínios Ativos acrescido da Combinação de Registradores com o método de Coloração de Grafos;
4. Proposta de uma solução superior para alocação global de registradores, embora ainda de maior custo comparada às técnicas tradicionais.

Capítulo 7

Trabalhos Futuros

A ordem de grandeza do algoritmo de Crescimento de Domínios Ativos não foi calculada nesta etapa do trabalho, devido ao fato de que não foi encontrada na literatura estudada a ordem de grandeza do algoritmo de alocação de registradores baseado em coloração de grafos. Desta maneira, apenas a ordem de grandeza do algoritmo CDA não seria uma medida aproveitável, pois não poderia ser comparada a ordem de grandeza da coloração de grafos. O cálculo da ordem de grandeza do algoritmo de Crescimento de Domínios Ativos foi, portanto, deixado como trabalho futuro.

Um trabalho futuro interessante também seria a otimização do Método de Alocação Global de Registradores Baseado no Crescimento de Domínios Ativos. Este método percorre o código do programa várias vezes: para se fazer a análise de Alcançabilidade e Consistência de Registradores, para emitir instruções não dependentes da solução de funções ϕ, para a solução de ϕ, e para a construção do grafo de dependência.

Uma outra contribuição importante seria a implementação deste método em um compilador mais robusto, como o gcc [1].

Bibliografia

[1] The gnu compiler collection project. http://gcc.gnu.org.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Company, 1979.

[3] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *PLDI89*, 1989.

[4] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, 1992.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *SIGPLAN89, SIGPLAN92*, 1990.

[6] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *Proceedings of the ACM SIGPLAN91*, 1991.

[7] G. J. Chaitin. Register allocation and spilling via graph coloring. *IBM Research*, 1982.

[8] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *Proceedings of the ACM SIGPLAN84*, 1984.

[9] Frederick Chow. Minimizing register usage penalty at procedure calls. *PLDI88*, 1988.

[10] Marcelo Silva Cintra. Alocação global de registradores de endereçamento usando cobertura do grafo de indexação e uma variação da forma ssa. Master's thesis, Universidade Estadual de Campinas, 2000.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Denneth Zadeck. An efficient method of computing static single assignment form. *Proceedings of the ACM SIGPLAN89*, 1989.

[12] Guilherme de Lima Ottoni. Alocação global de registradores de endereçamento para referências a vetores em dsps. Master's thesis, Universidade Estadual de Campinas, 2002.

[13] Richard A. Freiburghouse. Allocation via usage counts. *CACM*, 1974.

[14] Lal George and Andrew W. Appel. Iterated register coalescing. *Proceedings of the ACM TOPLAS96*, 1996.

[15] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. *PLDI89*, 1989.

[16] Glenn Holloway and Michael D. Smith. Machine suif, 2000. http://www.eecs.harvard.edu/hube/research/machsuif.html.

[17] Kathleen Knobe and F. Kenneth Zadeck. Register allocation using control trees. Technical report, Dept. of Comp. Sci., Brown Univ., Providence, RI, 1992.

[18] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[19] Guilherme Ottoni and Guido Araujo. Efficient array reference allocation for loops in embedded processors. *ESCODES02*, 2002.

[20] Guilherme Ottoni, Sandro Rigo, Rigo Araujo, Subramanian Rajagopalan, and Sharad Malik. Optimal live range merge for address register allocation in embedded programs. *CC2001*, 2001.

[21] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. *PLDI90*, 1990.

[22] Peter Steenkiste and John Hennessy. A simple interprocedual register allocation algorithm and its effectiveness for lisp. *ACM TOPLAS*, 1989.

[23] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. *Proceedings of the ACM SIGPLAN98*, 1998.

[24] David W. Wall. Global register allocation at link time. *SIGPLAN*, 1986.

[25] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

Apêndice A

Apêndice

A.1 *qsort* na linguagem intermediária do MachSUIF para a arquitetura Alpha

```
qsort:
        mov     $16,qsort.i
        mov     $17,qsort.j
        .loc    2 41
        ldil    $vr0,0
        mov     $vr0,qsort.pindex
        .loc    2 42
        mull    qsort.i,4,$vr2
        lda     $vr4,a
        mov     $vr4,$vr3
        mov     $vr3,$vr5
        addl    $vr5,$vr2,$vr6
        ldl     $vr7,0($vr6)
        mov     $vr7,$vr1
        mov     $vr1,qsort.first
        .loc    2 43
        ldil    $vr9,1
        addl    qsort.i,$vr9,$vr8
        mov     $vr8,qsort.k
        .loc    2 44
        ldil    $vr10,0
        mov     $vr10,qsort._quicksortTmp3
        ldil    $vr11,0
        cmpeq   qsort.pindex,$vr11,$vr84
        beq     $vr84,qsort._quicksortTmp4
        cmplt   qsort.j,qsort.k,$vr85
        bne     $vr85,qsort._quicksortTmp4
        ldil    $vr12,1
        mov     $vr12,qsort._quicksortTmp3
```

```
qsort._quicksortTmp4:
qsort._quicksortTmp1:
        beq     qsort._quicksortTmp3,qsort._quicksortTmp2
        mull    qsort.k,4,$vr14
        lda     $vr16,a
        mov     $vr16,$vr15
        mov     $vr15,$vr17
        addl    $vr17,$vr14,$vr18
        ldl     $vr19,0($vr18)
        mov     $vr19,$vr13
        cmple   $vr13,qsort.first,$vr86
        bne     $vr86,qsort._quicksortTmp12
        .loc    2 48
        mov     qsort.k,qsort.pindex
        br      qsort._quicksortTmp13
qsort._quicksortTmp12:
        mull    qsort.k,4,$vr21
        lda     $vr23,a
        mov     $vr23,$vr22
        mov     $vr22,$vr24
        addl    $vr24,$vr21,$vr25
        ldl     $vr26,0($vr25)
        mov     $vr26,$vr20
        cmple   qsort.first,$vr20,$vr87
        bne     $vr87,qsort._quicksortTmp14
        .loc    2 52
        mov     qsort.i,qsort.pindex
qsort._quicksortTmp14:
qsort._quicksortTmp13:
        .loc    2 54
        ldil    $vr28,1
        addl    qsort.k,$vr28,$vr27
        mov     $vr27,qsort.k
        .loc    2 44
        ldil    $vr29,0
        mov     $vr29,qsort._quicksortTmp3
        ldil    $vr30,0
        cmpeq   qsort.pindex,$vr30,$vr88
        beq     $vr88,qsort._quicksortTmp5
        cmplt   qsort.j,qsort.k,$vr89
        bne     $vr89,qsort._quicksortTmp5
        ldil    $vr31,1
        mov     $vr31,qsort._quicksortTmp3
qsort._quicksortTmp5:
        br      qsort._quicksortTmp1
qsort._quicksortTmp2:
        ldil    $vr32,0
        cmpeq   qsort.pindex,$vr32,$vr90
```

```
        bne     $vr90,qsort._quicksortTmp15
        .loc    2 59
        mull    qsort.pindex,4,$vr34
        lda     $vr36,a
        mov     $vr36,$vr35
        mov     $vr35,$vr37
        addl    $vr37,$vr34,$vr38
        ldl     $vr39,0($vr38)
        mov     $vr39,$vr33
        mov     $vr33,qsort.pivot
        .loc    2 62
        mov     qsort.i,qsort.l
        .loc    2 63
        mov     qsort.j,qsort.r
qsort._quicksortTmp6:
        .loc    2 67
        mull    qsort.l,4,$vr41
        lda     $vr43,a
        mov     $vr43,$vr42
        mov     $vr42,$vr44
        addl    $vr44,$vr41,$vr45
        ldl     $vr46,0($vr45)
        mov     $vr46,$vr40
        mov     $vr40,qsort.t
        .loc    2 68
        mull    qsort.r,4,$vr48
        lda     $vr50,a
        mov     $vr50,$vr49
        mov     $vr49,$vr51
        addl    $vr51,$vr48,$vr52
        ldl     $vr53,0($vr52)
        mov     $vr53,$vr47
        mull    qsort.l,4,$vr54
        lda     $vr56,a
        mov     $vr56,$vr55
        mov     $vr55,$vr57
        addl    $vr57,$vr54,$vr58
        stl     $vr47,0($vr58)
        .loc    2 69
        mull    qsort.r,4,$vr59
        lda     $vr61,a
        mov     $vr61,$vr60
        mov     $vr60,$vr62
        addl    $vr62,$vr59,$vr63
        stl     qsort.t,0($vr63)
qsort._quicksortTmp8:
        mull    qsort.l,4,$vr65
        lda     $vr67,a
```

```
        mov     $vr67,$vr66
        mov     $vr66,$vr68
        addl    $vr68,$vr65,$vr69
        ldl     $vr70,0($vr69)
        mov     $vr70,$vr64
        cmple   qsort.pivot,$vr64,$vr91
        bne     $vr91,qsort._quicksortTmp9
        .loc    2 71
        ldil    $vr72,1
        addl    qsort.l,$vr72,$vr71
        mov     $vr71,qsort.l
        br      qsort._quicksortTmp8
qsort._quicksortTmp9:
qsort._quicksortTmp10:
        mull    qsort.r,4,$vr74
        lda     $vr76,a
        mov     $vr76,$vr75
        mov     $vr75,$vr77
        addl    $vr77,$vr74,$vr78
        ldl     $vr79,0($vr78)
        mov     $vr79,$vr73
        cmplt   $vr73,qsort.pivot,$vr92
        bne     $vr92,qsort._quicksortTmp11
        .loc    2 73
        ldil    $vr81,1
        subl    qsort.r,$vr81,$vr80
        mov     $vr80,qsort.r
        br      qsort._quicksortTmp10
qsort._quicksortTmp11:
        cmple   qsort.l,qsort.r,$vr93
        bne     $vr93,qsort._quicksortTmp6
qsort._quicksortTmp7:
        .loc    2 76
        mov     qsort.l,qsort.k
        ldil    $vr83,1
        subl    qsort.k,$vr83,$vr82
        mov     qsort.i,$16
        mov     $vr82,$17
        lda     $27,qsort
        jsr     $26,($27)
        ldq     $gp,qsort.__gp_home__
        mov     qsort.k,$16
        mov     qsort.j,$17
        lda     $27,qsort
        jsr     $26,($27)
        ldq     $gp,qsort.__gp_home__
qsort._quicksortTmp15:
        .loc    2 81
```

```
        ret     ($26),1
        .end    qsort
        .text
        .align  4
        .align  4
        .globl  main
        .loc    2 0
        .ent    main
```

A.2 *bubsort* na linguagem intermediária do MachSUIF para a arquitetura Alpha

```
        # [target_lib: "alpha"]
        .sdata
        .align  4
        .align  0       # disable automatic alignment
_bubsortTmp0:
        .byte   83, 111, 114, 116, 101, 100, 58, 32, 37, 100, 0
        .file   2 "bubsort.c"
        .text
        .align  4
        .align  4
        .globl  main
        .loc    2 0
        .ent    main
(#NODO 0)
(#NODO 1)
main:
        .loc    2 24
        ldil    $vr0,1
        mov     $vr0,main.j
        .loc    2 26
        ldil    $vr1,20
        mov     $vr1,main.n
        .loc    2 27
        ldil    $vr2,0
        mov     $vr2,main.i
        .loc    2 29
        ldil    $vr3,12
        ldil    $vr4,0
        mull    $vr4,4,$vr5
        lda     $vr6,main.list
        mov     $vr6,$vr7
        addl    $vr7,$vr5,$vr8
        stl     $vr3,0($vr8)
        .loc    2 30
        ldil    $vr9,56
```

```
ldil    $vr10,1
mull    $vr10,4,$vr11
lda     $vr12,main.list
mov     $vr12,$vr13
addl    $vr13,$vr11,$vr14
stl     $vr9,0($vr14)
.loc    2 31
ldil    $vr15,34
ldil    $vr16,2
mull    $vr16,4,$vr17
lda     $vr18,main.list
mov     $vr18,$vr19
addl    $vr19,$vr17,$vr20
stl     $vr15,0($vr20)
.loc    2 32
ldil    $vr21,1
ldil    $vr22,3
mull    $vr22,4,$vr23
lda     $vr24,main.list
mov     $vr24,$vr25
addl    $vr25,$vr23,$vr26
stl     $vr21,0($vr26)
.loc    2 33
ldil    $vr27,6
ldil    $vr28,4
mull    $vr28,4,$vr29
lda     $vr30,main.list
mov     $vr30,$vr31
addl    $vr31,$vr29,$vr32
stl     $vr27,0($vr32)
.loc    2 34
ldil    $vr33,-1
ldil    $vr34,5
mull    $vr34,4,$vr35
lda     $vr36,main.list
mov     $vr36,$vr37
addl    $vr37,$vr35,$vr38
stl     $vr33,0($vr38)
.loc    2 35
ldil    $vr39,9
ldil    $vr40,6
mull    $vr40,4,$vr41
lda     $vr42,main.list
mov     $vr42,$vr43
addl    $vr43,$vr41,$vr44
stl     $vr39,0($vr44)
.loc    2 40
ldil    $vr45,13
```

```
        mov     $vr45,main.top
        ldil    $vr46,-1
(#NODO 2)
main._bubsortTmp1:
        ldil    $vr47,1
        cmple   main.top,$vr47,$vr100
        bne     $vr100,main._bubsortTmp2
        .loc    2 42
(#NODO 3)
        ldil    $vr48,1
        mov     $vr48,main.i
(#NODO 4)
main._bubsortTmp3:
        cmple   main.top,main.i,$vr101
        bne     $vr101,main._bubsortTmp4
(#NODO 5)
        mull    main.i,4,$vr50
        lda     $vr51,main.list
        mov     $vr51,$vr52
        addl    $vr52,$vr50,$vr53
        ldl     $vr54,0($vr53)
        mov     $vr54,$vr49
        ldil    $vr57,1
        addl    main.i,$vr57,$vr56
        mull    $vr56,4,$vr58
        lda     $vr59,main.list
        mov     $vr59,$vr60
        addl    $vr60,$vr58,$vr61
        ldl     $vr62,0($vr61)
        mov     $vr62,$vr55
        cmple   $vr49,$vr55,$vr102
        bne     $vr102,main._bubsortTmp5
(#NODO 6)
        .loc    2 46
        ldil    $vr65,1
        addl    main.i,$vr65,$vr64
        mull    $vr64,4,$vr66
        lda     $vr67,main.list
        mov     $vr67,$vr68
        addl    $vr68,$vr66,$vr69
        ldl     $vr70,0($vr69)
        mov     $vr70,$vr63
        mov     $vr63,main.temp
        .loc    2 47
        mull    main.i,4,$vr72
        lda     $vr73,main.list
        mov     $vr73,$vr74
        addl    $vr74,$vr72,$vr75
```

```
        ldl     $vr76,0($vr75)
        mov     $vr76,$vr71
        ldil    $vr78,1
        addl    main.i,$vr78,$vr77
        mull    $vr77,4,$vr79
        lda     $vr80,main.list
        mov     $vr80,$vr81
        addl    $vr81,$vr79,$vr82
        stl     $vr71,0($vr82)
        .loc    2 48
        mull    main.i,4,$vr83
        lda     $vr84,main.list
        mov     $vr84,$vr85
        addl    $vr85,$vr83,$vr86
        stl     main.temp,0($vr86)
(#NODO 7)
main._bubsortTmp5:
        .loc    2 50
        ldil    $vr88,1
        addl    main.i,$vr88,$vr87
        mov     $vr87,main.i
        br      main._bubsortTmp3
(#NODO 8)
main._bubsortTmp4:
        .loc    2 52
        ldil    $vr90,1
        subl    main.top,$vr90,$vr89
        mov     $vr89,main.top
        br      main._bubsortTmp1
(#NODO 9)
main._bubsortTmp2:
        lda     $vr91,_bubsortTmp0
        # [regs_used: "sparse", 3]
        ldil    $vr93,0
        mull    $vr93,4,$vr94
        lda     $vr95,main.list
        mov     $vr95,$vr96
        addl    $vr96,$vr94,$vr97
        ldl     $vr98,0($vr97)
        mov     $vr98,$vr92
        mov     $vr91,$16
        mov     $vr92,$17
        lda     $27,printf
        jsr     $26,($27)
        ldq     $gp,main.__gp_home__
        .loc    2 63
        ldil    $vr99,0
        mov     $vr99,$0
```

```
        ret     ($26),1
        .end    main
(#NODO 10)
```

A.3 Código do *bubsort* resultante da primeira implementação do alocador de registradores baseado em crescimento de domínios ativos e combinação de registradores

```
        # [target_lib: "alpha"]
        .sdata
        .align  4
        .align  0       # disable automatic alignment
_bubsortTmp0:
        .byte   83, 111, 114, 116, 101, 100, 58, 32, 37, 100, 0
        .file   2 "bubsort.c"
        .text
        .align  4
        .align  4
        .globl  main
        .loc    2 0
        .ent    main
main:
        .loc    2 24
        ldil    $20,1
        .loc    2 26
        ldil    $21,7
        .loc    2 27
        ldil    $1,0
        .loc    2 29
        ldil    $1,12
        ldil    $2,0
        mull    $2,4,$2
        lda     $3,main.list
        addl    $3,$2,$3
        stl     $1,0($3)
        .loc    2 30
        ldil    $3,56
        ldil    $4,1
        mull    $4,4,$4
        lda     $5,main.list
        addl    $5,$4,$5
        stl     $3,0($5)
        .loc    2 31
        ldil    $5,34
        ldil    $6,2
        mull    $6,4,$6
```

```
        lda     $7,main.list
        addl    $7,$6,$7
        stl     $5,0($7)
        .loc    2 32
        ldil    $7,1
        ldil    $8,3
        mull    $8,4,$8
        lda     $22,main.list
        addl    $22,$8,$22
        stl     $7,0($22)
        .loc    2 33
        ldil    $22,6
        ldil    $16,4
        mull    $16,4,$16
        lda     $17,main.list
        addl    $17,$16,$17
        stl     $22,0($17)
        .loc    2 34
        ldil    $17,-1
        ldil    $18,5
        mull    $18,4,$18
        lda     $19,main.list
        addl    $19,$18,$19
        stl     $17,0($19)
        .loc    2 35
        ldil    $19,9
        ldil    $20,6
        mull    $20,4,$21
        lda     $20,main.list
        addl    $20,$21,$21
        stl     $19,0($21)
        .loc    2 40
        ldil    $21,7
        ldil    $20,-1
main._bubsortTmp1:
        ldil    $0,1
        cmple   $21,$0,$0
        bne     $0,main._bubsortTmp2
        .loc    2 42
        ldil    $1,1
main._bubsortTmp3:
        cmple   $21,$1,$0
        bne     $0,main._bubsortTmp4
        mull    $1,4,$2
        lda     $3,main.list
        addl    $3,$2,$3
        ldl     $4,0($3)
        ldil    $5,1
```

```
        addl    $1,$5,$5
        mull    $5,4,$0
        lda     $5,main.list
        addl    $5,$0,$6
        ldl     $6,0($6)
        stl     $1,main.i
        cmple   $4,$6,$1
        bne     $1,main._bubsortTmp5
        .loc    2 46
        ldil    $2,1
        ldl     $1,main.i
        addl    $1,$2,$3
        mull    $3,4,$3
        lda     $4,main.list
        addl    $4,$3,$5
        ldl     $5,0($5)
        mov     $5,$0
        .loc    2 47
        mull    $1,4,$5
        lda     $6,main.list
        addl    $6,$5,$6
        stl     $1, main.i
        ldl     $1,0($6)
        mov     $1,$2
        ldil    $3,1
        ldl     $1,main.i
        addl    $1,$3,$3
        mull    $3,4,$4
        lda     $5,main.list
        addl    $5,$4,$5
        stl     $2,0($5)
        .loc    2 48
        mull    $1,4,$5
        lda     $6,main.list
        addl    $6,$5,$6
        stl     $0,0($6)
main._bubsortTmp5:
        .loc    2 50
        ldil    $6,1
        addl    $1,$6,$1
        br      main._bubsortTmp3
main._bubsortTmp4:
        .loc    2 52
        ldil    $20,1
        subl    $21,$20,$21
        br      main._bubsortTmp1
main._bubsortTmp2:
        lda     $16,_bubsortTmp0
```

```
        # [regs_used: "sparse", 3]
        ldil    $6,0
        mull    $6,4,$6
        lda     $7,main.list
        addl    $7,$6,$7
        ldl     $17,0($7)
        lda     $27,printf
        jsr     $26,($27)
        ldq     $gp,main.__gp_home__
        .loc    2 63
        ldil    $0,0
        ret     ($26),1
        .end    main
```

A.4 Código do *bubsort* resultante da segunda implementação do alocador de registradores baseado em crescimento de domínios ativos e combinação de registradores

```
        # [target_lib: "alpha"]
        .sdata
        .align  4
        .align  0       # disable automatic alignment
_bubsortTmp0:
        .byte   83, 111, 114, 116, 101, 100, 58, 32, 37, 100, 0
        .file   2 "bubsort.c"
        .text
        .align  4
        .align  4
        .globl  main
        .loc    2 0
        .ent    main
main:
        .loc    2 24
        ldil    $21,1
        .loc    2 26
        ldil    $19,7
        .loc    2 27
        ldil    $1,0
        .loc    2 29
        ldil    $1,12
        ldil    $2,0
        mull    $2,4,$2
        lda     $3,main.list
        addl    $3,$2,$3
        stl     $1,0($3)
        .loc    2 30
```

```
        ldil    $3,56
        ldil    $4,1
        mull    $4,4,$4
        lda     $5,main.list
        addl    $5,$4,$5
        stl     $3,0($5)
        .loc    2 31
        ldil    $5,34
        ldil    $6,2
        mull    $6,4,$6
        lda     $7,main.list
        addl    $7,$6,$7
        stl     $5,0($7)
        .loc    2 32
        ldil    $7,1
        ldil    $8,3
        mull    $8,4,$8
        lda     $22,main.list
        addl    $22,$8,$22
        stl     $7,0($22)
        .loc    2 33
        ldil    $22,6
        ldil    $16,4
        mull    $16,4,$16
        lda     $17,main.list
        addl    $17,$16,$17
        stl     $22,0($17)
        .loc    2 34
        ldil    $17,-1
        ldil    $18,5
        mull    $18,4,$18
        lda     $19,main.list
        addl    $19,$18,$19
        stl     $17,0($19)
        .loc    2 35
        ldil    $20,9
        ldil    $21,6
        mull    $21,4,$19
        lda     $21,main.list
        addl    $21,$19,$19
        stl     $20,0($19)
        .loc    2 40
        ldil    $19,7
        ldil    $21,-1
main._bubsortTmp1:
        ldil    $21,1
        cmple   $19,$21,$21
        bne     $21,main._bubsortTmp2
```

```
        .loc    2 42
        ldil    $7,1
main._bubsortTmp3:
        cmple   $19,$7,$21
        bne     $21,main._bubsortTmp4
        mull    $7,4,$21
        lda     $0,main.list
        addl    $0,$21,$1
        ldl     $2,0($1)
        ldil    $3,1
        addl    $7,$3,$4
        mull    $4,4,$21
        lda     $4,main.list
        addl    $4,$21,$5
        ldl     $6,0($5)
        cmple   $2,$6,$6
        bne     $6,main._bubsortTmp5
        .loc    2 46
        ldil    $21,1
        addl    $7,$21,$0
        mull    $0,4,$1
        lda     $2,main.list
        addl    $2,$1,$3
        ldl     $4,0($3)
        mov     $4,$21
        .loc    2 47
        mull    $7,4,$4
        lda     $5,main.list
        addl    $5,$4,$6
        ldl     $6,0($6)
        mov     $6,$0
        ldil    $1,1
        addl    $7,$1,$2
        mull    $2,4,$3
        lda     $4,main.list
        addl    $4,$3,$4
        stl     $0,0($4)
        .loc    2 48
        mull    $7,4,$5
        lda     $6,main.list
        addl    $6,$5,$6
        stl     $21,0($6)
main._bubsortTmp5:
        .loc    2 50
        ldil    $6,1
        addl    $7,$6,$7
        br      main._bubsortTmp3
main._bubsortTmp4:
```

```
        .loc    2 52
        ldil    $21,1
        subl    $19,$21,$19
        br      main._bubsortTmp1
main._bubsortTmp2:
        lda     $16,_bubsortTmp0
        # [regs_used: "sparse", 3]
        ldil    $6,0
        mull    $6,4,$6
        lda     $7,main.list
        addl    $7,$6,$7
        ldl     $17,0($7)
        lda     $27,printf
        jsr     $26,($27)
        ldq     $gp,main.__gp_home__
        .loc    2 63
        ldil    $0,0
        ret     ($26),1
        .end    main
```

A.5 Código do *bubsort* resultante da terceira implementação do alocador de registradores baseado em crescimento de domínios ativos e combinação de registradores

```
        # [target_lib: "alpha"]
        .sdata
        .align  4
        .align  0       # disable automatic alignment
_bubsortTmp0:
        .byte   83, 111, 114, 116, 101, 100, 58, 32, 37, 100, 0
        .file   2 "bubsort.c"
        .text
        .align  4
        .align  4
        .globl  main
        .loc    2 0
        .ent    main
main:
        .loc    2 24
        ldil    $21,1
        .loc    2 26
        ldil    $19,7
        .loc    2 27
        ldil    $1,0
        .loc    2 29
```

```
ldil    $1,12
ldil    $2,0
mull    $2,4,$2
lda     $3,main.list
addl    $3,$2,$3
stl     $1,0($3)
.loc    2 30
ldil    $3,56
ldil    $4,1
mull    $4,4,$4
lda     $5,main.list
addl    $5,$4,$5
stl     $3,0($5)
.loc    2 31
ldil    $5,34
ldil    $6,2
mull    $6,4,$6
lda     $7,main.list
addl    $7,$6,$7
stl     $5,0($7)
.loc    2 32
ldil    $7,1
ldil    $8,3
mull    $8,4,$8
lda     $22,main.list
addl    $22,$8,$22
stl     $7,0($22)
.loc    2 33
ldil    $22,6
ldil    $16,4
mull    $16,4,$16
lda     $17,main.list
addl    $17,$16,$17
stl     $22,0($17)
.loc    2 34
ldil    $17,-1
ldil    $18,5
mull    $18,4,$18
lda     $19,main.list
addl    $19,$18,$19
stl     $17,0($19)
.loc    2 35
ldil    $20,9
ldil    $21,6
mull    $21,4,$19
lda     $21,main.list
addl    $21,$19,$19
stl     $20,0($19)
```

```
        .loc    2 40
        ldil    $19,7
        ldil    $21,-1
main._bubsortTmp1:
        ldil    $21,1
        cmple   $19,$21,$21
        bne     $21,main._bubsortTmp2
        .loc    2 42
        ldil    $7,1
main._bubsortTmp3:
        cmple   $19,$7,$21
        bne     $21,main._bubsortTmp4
        mull    $7,4,$21
        lda     $0,main.list
        addl    $0,$21,$1
        ldl     $2,0($1)
        ldil    $3,1
        addl    $7,$3,$4
        mull    $4,4,$21
        lda     $4,main.list
        addl    $4,$21,$5
        ldl     $6,0($5)
        cmple   $2,$6,$6
        bne     $6,main._bubsortTmp5
        .loc    2 46
        ldil    $21,1
        addl    $7,$21,$0
        mull    $0,4,$1
        lda     $2,main.list
        addl    $2,$1,$3
        ldl     $4,0($3)
        mov     $4,$21
        .loc    2 47
        mull    $7,4,$4
        lda     $5,main.list
        addl    $5,$4,$6
        ldl     $6,0($6)
        mov     $6,$0
        ldil    $1,1
        addl    $7,$1,$2
        mull    $2,4,$3
        lda     $4,main.list
        addl    $4,$3,$4
        stl     $0,0($4)
        .loc    2 48
        mull    $7,4,$5
        lda     $6,main.list
        addl    $6,$5,$6
```

```
        stl     $21,0($6)
main._bubsortTmp5:
        .loc    2 50
        ldil    $6,1
        addl    $7,$6,$7
        br      main._bubsortTmp3
main._bubsortTmp4:
        .loc    2 52
        ldil    $21,1
        subl    $19,$21,$19
        br      main._bubsortTmp1
main._bubsortTmp2:
        lda     $16,_bubsortTmp0
        # [regs_used: "sparse", 3]
        ldil    $6,0
        mull    $6,4,$6
        lda     $7,main.list
        addl    $7,$6,$7
        ldl     $17,0($7)
        lda     $27,printf
        jsr     $26,($27)
        ldq     $gp,main.__gp_home__
        .loc    2 63
        ldil    $0,0
        ret     ($26),1
        .end    main
```