

Alocação de Registradores a Dado de Tamanho Variável e Escalonamento de Instruções

Mariza Andrade da Silva Bigonha¹
José Lucas Mourão Rangel Netto²

Resumo

Em diversas arquiteturas existentes, i860, RS2000, M88000, etc., é comum que candidatos a alocação de registradores tenham necessidades diferentes em relação ao tamanho dos registradores, dependendo da precisão do tipo de dados que eles representem. Por exemplo, pares de registradores são muitas vezes necessários em processadores para representar duas metades de registradores usando precisão dupla para cargas, armazenamentos e movimentação de registradores. Levando em conta estas necessidades, o problema da alocação de registradores para processadores superescalares é encontrar um algoritmo ótimo tal que ele faça uso de um número mínimo de registradores físicos, que não derrame valores ainda vivos para a memória e cujo grafo de escalonamento não possua dependências falsas.

Este artigo mostra como modificar o algoritmo de Pinter para tratar da necessidade por tamanhos diferentes de registradores usando uma única estrutura de dados que embute informações sobre restrições de escalonamento e efetuar a alocação de registradores sem perder as propriedades delineadas no parágrafo anterior.

Abstract

Several superscalar processors, Intel i860, RS2000, Sparc, Motorola 88000, etc., depending on the data they represent, have different needs with respect to register size. For instance, candidates to register allocation may need register pairs to represent two halves using double precision in some instructions like load, store and moves between instructions. Thus, taking into account these facts, the problem of register allocation for this class of machines is to find an optimal register allocation, which uses a minimum number of register, that minimize spilling of live values to memory and whose scheduling graph does not have false dependence.

This paper shows how to modify the Pinter's algorithm to manage the need for different size registers and using a unique data structure to represent the scheduling constraints to perform a register allocation without losing the properties presented in the last paragraph.

¹MSc (UFMG - 1985), DSc (PUC/RJ - 1994). É professora do DCC - UFMG. Áreas de interesse: linguagens de programação, compiladores, arquitetura. E-mail: mariza@dcc.ufmg.br

²MSc Eng. Elétrica (COPPE/UF RJ), Ph.D. em Ciência da Computação (Wisconsin, E.U.A.). É professor do Inst. de Matemática da UFRJ e do DI da PUC/RJ. Áreas de interesse: engenharia de software, compiladores, linguagens de programação, teoria da computação. E-mail: rangel@inf.puc-rio.br

O modelo mais usado nos algoritmos [12], [9], [10], [11]. Dado um conjunto de instruções que estão aptas para serem executadas, seleciona-se uma heurística para selecionar a próxima instrução para a lista. Em geral, esta abordagem é baseada no custo, onde se refere-se ao número de ar-

todo o paralelismo do programa pode ser explorado. A desvantagem é que ela aumenta o tempo entre a escrita em um registrador e sua posterior leitura, e pode provocar derramamento de alguns registradores. Essa desvantagem é a principal razão porque alguns pesquisadores preferem efetuar o escalonamento após a alocação. Por outro lado, o escalonamento de instruções após a alocação torna-se restrito porque o alocador de registradores pode, inadvertidamente, introduzir dependências, atribuindo o mesmo registrador para instruções sem nenhuma relação. Na presença de pequenos blocos básicos, esta restrição não afeta muito o desempenho do sistema, mas para blocos básicos maiores e *pipelines* longas esta restrição pode acarretar uma grande perda no desempenho [11].

Além disto, a complexidade do problema devido à ausência de comunicação entre a alocação de registradores e o escalonamento depende da interação entre o conjunto de instruções e das características do processador, tais como, as latências das instruções e o tamanho do conjunto de instruções. Na maioria das máquinas superescalares, a latência da instrução *load* é de 2 a 3 ciclos de máquina e grande parte das operações com inteiros possuem latência de 1 ciclo. Portanto, a ausência de comunicação entre estas duas funções em programas que contêm um grande percentual de operações com inteiros conduz a uma degradação mínima do sistema. Em contrapartida, nos programas científicos este problema é significativo: (1) programas científicos possuem um grau mais elevado de paralelismo entre instruções em função de um maior número de blocos básicos possuindo muitas expressões independentes. (2) estes programas utilizam muitas operações de ponto flutuante. Como estas operações, em geral, gastam vários ciclos, isto acarreta longos atrasos nas *pipelines*. (3) blocos muito grandes conduzem a muitos *pseudo-registradores* locais e globais.

2 Revisão da Literatura

2.1 Algoritmos de Escalonamento

Muitos processadores, tais como CDC 6600 e IBM 360/90, efetuam escalonamento de instruções dinamicamente no *hardware*. Mais recentemente, compiladores para processadores RISC têm utilizado técnicas de escalonamento de instruções no código de máquina. Os algoritmos de escalonamento considerados "ótimos", exceto em raríssimos casos, são NP-completos, mesmo não levando em consideração os recursos computacionais e restrições de registradores [2]. Consequentemente, heurísticas são utilizadas com frequência. Bernstein e Gertner apresentam um algoritmo, considerado de tempo polinomial, para o caso onde uma única *pipeline* possui um atraso máximo de um ciclo. Proebsting e Fischer apresentam um algoritmo, considerado de tempo linear, que minimiza o tempo de execução e o uso de registradores locais, dada uma árvore de expressão e supondo uma máquina onde a latência de todas as instruções é de 1 ciclo, exceto pela instrução *load*, cuja latência é de 2 ciclos [18].

O modelo mais usado nos algoritmos de escalonamento é a *lista de escalonamento* [8], [5], [20], [12], [9], [10], [11]. Dado um grafo de escalonamento, o escalonador mantém uma lista de instruções que estão aptas para serem escalonadas sem provocar atrasos. A cada iteração utiliza-se uma heurística para selecionar uma instrução pronta para ser escalonada e então atualiza-se a lista. Em geral, esta abordagem possui, no pior caso, um tempo de execução equivalente a $O(e)$, onde e refere-se ao número de arestas no grafo, mas a heurística pode crescer em complexidade.

Em uma *lista de escalonamento* escolhe-se o vértice que possui a prioridade mais elevada em

relação aos outros na lista. A heurística mais utilizada para atribuir prioridades denomina-se *distância máxima* ou *peso do vértice* e é definida como o comprimento do caminho mais longo através do grafo de escalonamento, partindo do vértice da instrução até o vértice folha. O comprimento de um caminho é a soma de todos os valores de rótulos das arestas ao longo do caminho. O raciocínio utilizado é que o vértice mais longe de ser atingido é o mais crítico, vértices menos importantes podem ser escalonados mais tarde. A *distância máxima* é considerada uma heurística inicial relativamente satisfatória, mas existem situações onde heurísticas alternativas ou complementares são mais vantajosas. Por exemplo, aquela que atribui uma prioridade mais alta para *vértices possuindo mais sucessores*. A idéia por trás desta heurística é que escalonando um vértice com mais sucessores, criam-se mais oportunidades para o escalonador nos ciclos subseqüentes e permite-se que mais vértices estejam prontos para serem escalonados mais cedo. Uma outra heurística atribui uma prioridade mais alta ao *vértice que possui uma latência maior em relação a um de seus sucessores*. O ato de escalonar tal vértice primeiro permite mais oportunidades de preencher o período de latência com o escalonamento de outras instruções.

O algoritmo *Rank* [16] é uma alternativa como um segundo nível de prioridade. Ele incorpora a *distância máxima* e o *número de sucessores* em um único grau de hierarquia. O algoritmo atribui a todas as folhas um grau hierárquico alto, por exemplo, o número de vértices do grafo de escalonamento mais o total dos rótulos de todas as arestas. A diminuição do grau é efetuada das folhas para a raiz. Para atribuir o grau ao vértice *i*, cada um de seus sucessores é colocado em uma tentativa de escalonamento, baseada em seu grau hierárquico, e então, é atribuído ao vértice *i* o grau que reflete sua posição no escalonamento em relação aos seus sucessores.

Algumas *listas de escalonamento* atualizam as prioridades durante o processo de escalonamento. Por exemplo, Warren [20] constrói seu escalonador para o sistema RS/6000 baseado no método de Auslander e Hopkins [15] com algumas heurísticas que limitam o efeito do escalonamento sobre o alocador de registradores. Em particular, Warren reduz a prioridade das cargas (*loads*) à medida que a necessidade por registradores aumenta.

Outros algoritmos, diferentes daqueles que utilizam *lista de escalonamento*, também são encontrados na literatura. Landskov et al. [14] utiliza o algoritmo do *Caminho Crítico* para escalonar todos os vértices que estão no caminho mais longo, desde a raiz do grafo até a folha. Outro algoritmo, também descrito por Landskov et al., *Separação e Avaliação* (*branch and bound*), constrói uma árvore contendo instruções que podem vir a ser escalonadas. Cada folha representa um escalonamento completo para o grafo de código.

2.2 Alocação de Registradores

O método mais utilizado para alocação de registradores, coloração de grafos, é baseado no trabalho de Chaitin [7]. Coloração de grafos modela o problema da alocação de registradores como um grafo de interferência (Seção 1). O método usado para colorir grafos, atribuindo cores diferentes para vértices que se interferem, é análogo a atribuir registradores físicos para *pseudo-registradores*.

Uma coloração do grafo de interferência na qual não existem dois vértices incidentes na mesma aresta que possuam a mesma cor induz uma alocação de registradores⁵ na qual todo valor permanece em um registrador enquanto ele estiver vivo. Diz-se que tal coloração do grafo de interferência é ótima e produz ótima alocação de registradores se ele usar o menor conjunto de

⁵se o número de registradores na máquina não é menor do que o número de cores

cores do total de cores disponíveis.

No método de Chaitin [7], todos os valores que podem ser colocados em registradores de código. O alocador de registradores atribui *pseudo-registradores* físicos por um período necessário. Tanto neste método de coloração de grafos quanto no método de *pseudo-registradores* são atribuídos durante todo o tempo de execução.

A principal característica da abordagem de Chaitin é que o alocador é capaz de identificar a alocação global de registradores e atribuir *pseudo-registradores* para a alocação dentro de um período necessário. *Pseudo-registrador* é atribuído a

2.3 Estratégias de Geração de Código

Muitos compiladores para arquitetura de computadores separam o escalonamento de instruções separadamente das fases de geração de código e RASE, onde cada uma representa

• POSTPASS

O método de Hennessy e Gross [11] atribui *pseudo-registradores* antes do escalonamento e utiliza *listas de escalonamento* para atualizar a alocação de *pseudo-registradores* físicos que são usados em cada nó. O método de Gibbons e Muchnik [12] diferencia a alocação entre as duas computações, a alocação de nós no DAG de código. O método de Hennessy e Gross produzindo, na maioria das vezes, a reordenação aumenta a complexidade da alocação.

• IPS (Integrated Prepass Scheduler)

No método de Goodman e Hsu [11], a alocação de *pseudo-registradores*, mas esta estratégia pode estar simultaneamente ativa para o número de *pseudo-registradores* existentes entre as instruções.

O algoritmo básico de escalonamento de *pseudo-registradores* locais ativos no escalonador está no limite, ele tenta atribuir um valor a cada *pseudo-registrador* e, caso ele não consiga, pode exceder o número de *pseudo-registradores* locais. São inseridos caso o escalonador tente

⁶do inglês: spilling

cores do total de cores disponíveis.

No método de Chaitin [7], todas as expressões temporárias, subexpressões comuns e variáveis que podem ser colocadas em registradores são atribuídas a *pseudo-registradores* durante a seleção de código. O alocador de registradores considera todos os *pseudo-registradores*, locais e globais. Ele aloca registradores físicos para alguns derramando⁶ para a memória uns valores quando necessário. Tanto neste método como nos métodos de Briggs et al. [6] e Bernstein et al. [1], os *pseudo-registradores* são atribuídos a um registrador físico ou são derramados para a memória durante todo o tempo de execução do bloco.

A principal característica da abordagem de Chaitin é que ela é uniforme e sistemática. O alocador é capaz de identificar a alocação dentro de blocos básicos como um caso particular da alocação global de registradores sem a necessidade de especificar um sub-conjunto de registradores para a alocação dentro dos mesmos. Uma das desvantagens deste modelo é que um *pseudo-registrador* é atribuído a um registrador físico ou derramado por todo seu tempo de vida.

2.3 Estratégias de Geração e Otimização de Código

Muitos compiladores para arquiteturas superescalares efetuam a alocação de registradores e o escalonamento de instruções separadamente, ignorando a necessidade de algum tipo de integração entre estas duas fases. Esta seção mostra 3 estratégias de geração de código: POSTPASS, IPS e RASE, onde cada uma representa um nível de comunicação diferente entre estas fases.

• POSTPASS

O método de Hennessy e Gross [12] e o método de Gibbons e Muchnick [10] efetuam a alocação de registradores antes do escalonamento de instruções nos blocos básicos. Ambos os escalonadores utilizam *listas de escalonamento* acrescidas de heurísticas para a escolha entre os candidatos ao escalonamento. A diferença marcante entre estes dois métodos está no manuseio dos registradores físicos que são usados em duas computações diferentes, dentro de um mesmo bloco. O método de Gibbons e Muchnick adiciona arestas ao DAG de código para forçar uma ordenação entre as duas computações. O escalonador é $O(n^2)$, onde n refere-se ao número de nós no DAG de código. O método de Hennessy e Gross permite ao escalonador reordenar as computações, produzindo, na maioria das vezes, um código de melhor qualidade. Contudo, a reordenação aumenta a complexidade do escalonamento para $O(n^4)$.

• IPS (Integrated Prepass Scheduling)

No método de Goodman e Hsu [11], o escalonamento de instruções precede a alocação de registradores, mas esta estratégia pré-estabelece um número de *pseudo-registradores* locais que podem estar simultaneamente ativos em cada bloco básico. O fato de estabelecer um limite para o número de *pseudo-registradores* locais acarreta a perda da possibilidade de explorar o paralelismo existente entre as instruções.

O algoritmo básico de escalonamento é similar àquele de POSTPASS, exceto quando o número de *pseudo-registradores* locais ativos é maior ou igual ao limite pré-estabelecido. Enquanto o escalonador está no limite, ele tenta escalonar uma instrução que libera um *pseudo-registrador*, e, caso ele não consiga, pode exceder o limite. Após o escalonamento, o alocador atribui registradores aos *pseudo-registradores* locais dentro dos blocos básicos. Derramamentos para a memória são inseridos caso o escalonador tenha excedido o seu limite de registradores. Goodman e Hsu

⁶do ingles: spilling

atribuem registradores físicos para *pseudo-registradores* globais considerados importantes, como por exemplo, variáveis de indução e invariantes de *loops*, antes do escalonamento de instruções.

• RASE (*Register Allocation with Schedule Estimate*)

A estratégia RASE [4] fornece ao alocador de registradores estimativas sobre custos de escalonamento que permite ao alocador avaliar os efeitos de suas escolhas no escalonador. Como nas outras estratégias, um seletor de instruções gera um código de máquina simples que define e utiliza *pseudo-registradores*. E então, o controle passa para a estratégia. Existem dois aspectos na solução usada em RASE: a responsabilidade pela alocação de registradores é distribuída e é feito um pré-escalonamento.

RASE é composta de 3 fases: (1) PRESCHED (pré escalonamento): calcula a função do custo do escalonamento para cada bloco. Dado um limite de registradores, esta função retorna uma *estimativa de custo do escalonamento*. Uma *estimativa de custo de escalonamento* é o número estimado de ciclos de máquina necessários para executar as instruções enquanto continua dentro do limite de registradores pré-estabelecido.

(2) GRA: aloca registradores somente para *pseudo-registradores* globais. Ele utiliza a *estimativa de custo do escalonamento* para calcular o custo incremental reduzindo o limite de registradores. O *custo incremental* representa os ciclos adicionais necessários para escalonar um bloco com redução do paralelismo entre as instruções.

(3) FINALSCHED: escalona cada bloco atribuindo registradores físicos aos *pseudo-registradores* locais de acordo com o limite de registradores para ele calculado. O algoritmo do escalonador propriamente dito denomina-se SCHED.

2.4 Análise Crítica das Estratégias POSTPASS, IPS e RASE

Os dados estatísticos apresentados nesta seção fazem parte da experiência realizada com o sistema Marion [3].

Bradlee emulou três arquiteturas baseadas em um modelo de máquina que corresponde a classe das arquiteturas superescalares para realizar as experiências sobre o efeito do desempenho de processadores superescalares em relação ao tamanho do conjunto de registradores, sua organização e sobre as latências das operações usando estratégias de geração de código distintas. A maior parte das características destas máquinas foram adaptadas de MC88000 e MIPS R2000/R2010. O objetivo foi escolher componentes que poderiam constituir um modelo básico razoável de RISC e então variar características dos modelos. Ao variar uma em particular, outros aspectos da arquitetura permanecem inalterados. As características comuns são: (1) CPU: contém uma *pipeline* de 4 estágios para operações de inteiros, exceto multiplicação e divisão. (2) FPU: contém *pipelines* separadas para adição e multiplicação que compartilham os estágios inicial e final da *pipeline* de adição. (3) Barramento de 32 bits: conecta a CPU ao *cache* de memória e dados. (4) Operações *load*: usam uma *pipeline* de *load* de 2 estágios. (5) Cargas de ponto flutuante de precisão simples e cargas de inteiros: latência de 2 ciclos. (6) Cargas de precisão dupla: 3 ciclos. (7) Organização dos registradores: compartilhada e separada. Na versão separada, registradores são divididos em 2 partições iguais, para valores inteiros e de ponto flutuante. Experiências realizadas comparam dois conjuntos de latências de operações: *longerop* e *shorterop*. *Longerop* modela latências da MC88000, e *shorterop* modela latências da RS2000. Para os *loads*, *longerop* modela máquinas sem *cache* como o Cray-1 e *shorterop* modela máquinas com *cache*.

Cinco parâmetros são examinados: (1) estratégia de geração de código, (2) tamanho do conjunto de registradores, (3) organização dos registradores, (4) latência das operações, (5) latência de

instruções *load*.

Resultados obtidos demonstraram, mitigando o uso de registradores e registradores deve operar. Com *registradores* globais referenciados registradores. Conseqüentemente, evita significantes derramam alocação, o escalonador não precis Além disto, a ordem das instruções final, com isto o alocador tem um:

Vantagens de RASE sobre POSTPASS escalonamento em RASE faz com que entre as necessidades por registradores básico um número suficiente das instruções, mas não o bastante para RASE atrasa a alocação até o escalonador registradores específicos.

Nas três arquiteturas examinadas, média, exceto em programas com registradores ou latências maiores, RASE

Em relação ao tempo de compilação, mais que o dobro que POSTPASS deve verificar os limites de registradores escalona cada bloco três ou quatro vezes a arquitetura e porque o algoritmo

IPS e RASE produzem código 12% hipótese de que a falta de comunicações ineficiente. Por outro lado, idêntico. RASE é melhor que IPS básicos. Contudo, para as três arquiteturas existente em RASE não é necessária a melhor escolha, porque além de simplificar a compilação. Estratégias como RASE blocos básicos, característica comum

Duas modificações podem ser inseridas no código:

- (1) Alocação global de registradores: res uma ordenação computada por mais parecida com IPS, contudo RASE de registradores locais e além disto.
- (2) Código derramado: RASE por PRESCHED, o que eliminaria derramamento estas duas funções juntas.

⁷Escalonamento feito antes da alocação

instruções *load*.

Resultados obtidos demonstraram que IPS é melhor que POSTPASS por duas razões: (1) limitando o uso de registradores locais, IPS emula as restrições sobre as quais o alocador de registradores deve operar. Como o limite de registradores leva em consideração os *pseudo-registradores* globais referenciados dentro do bloco, isto impede o crescimento da pressão por registradores. Conseqüentemente, reduz o derramamento de uma forma geral, e, em particular, evita significantes derramamentos em blocos importantes. (2) Como IPS escalona antes da alocação, o escalonador não precisa se preocupar com as restrições de registradores específicos. Além disto, a ordem das instruções apresentadas ao alocador é mais próxima do escalonamento final, com isto o alocador tem uma visão melhor da interferência entre *pseudo-registradores*.

Vantagens de RASE sobre POSTPASS são atribuídas a dois fatores: (1) estimativa do custo de escalonamento em RASE faz com que o alocador de registrador encontre um ponto de equilíbrio entre as necessidades por registradores locais e globais. Conseqüentemente, fornece ao bloco básico um número suficiente dos mesmos para explorar o paralelismo existente entre as instruções, mas não o bastante para provocar derramamentos desnecessários em algum ponto. (2) RASE atrasa a alocação até o escalonamento final, reduzindo assim, restrições devido ao uso de registradores específicos.

Nas três arquiteturas examinadas, RASE produz uma pequena aceleração sobre IPS, 1% na média, exceto em programas possuindo grandes blocos básicos. Arquiteturas com poucos registradores ou latências maiores, RASE obtém acelerações maiores.

Em relação ao tempo de compilação, IPS e RASE gastam mais tempo que POSTPASS. IPS gasta mais que o dobro que POSTPASS porque ele escalona cada bloco duas vezes e o escalonador deve verificar os limites de registradores. RASE gasta três vezes mais tempo que IPS porque ele escalona cada bloco três ou quatro vezes, dependendo do número do conjunto de registradores da arquitetura e porque o algoritmo de escalonamento de instruções é mais complicado que IPS.

IPS e RASE produzem código 12% em média mais rápido que POSTPASS, reafirmando a hipótese de que a falta de comunicação entre a alocação e o escalonamento resulta em um escalonamento ineficiente. Por outro lado, o desempenho do código gerado por IPS e RASE é quase idêntico. RASE é melhor que IPS somente em alguns programas que possuem grandes blocos básicos. Contudo, para as três arquiteturas utilizadas na experiência, o acoplamento acirrado existente em RASE não é necessário. Portanto, dentre as três estratégias, IPS é considerada a melhor escolha, porque além de ser conceitualmente mais simples, é mais barata em tempo de compilação. Estratégias como RASE podem ser uma boa opção para programas com grandes blocos básicos, característica comum nas novas técnicas *loop unrolling* e *trace scheduling* [13].

Duas modificações podem ser inseridas em RASE para melhorar a qualidade do código produzido:

- (1) Alocação global de registradores: RASE poderia apresentar ao alocador global de registradores uma ordenação computada por PRESCHED. A reordenação *prepass*⁷ poderia tornar RASE mais parecida com IPS, contudo RASE reteria heurísticas mais precisas para determinar o limite de registradores locais e além disto, ele poderia reduzir o derramamento de uma forma geral.
- (2) Código derramado: RASE poderia separar a alocação local de registradores de FINALSCCHED, o que eliminaria derramamentos locais desnecessários, oriundos da dificuldade de efetuar estas duas funções juntas.

⁷ Escalonamento feito antes da alocação de registradores.

3 Alocação de Registradores a Tipo de Tamanho Variável e Escalonamento de Instruções

Esta nova abordagem baseia-se nos algoritmos propostos por [17] e endereça deficiências dos modelos apresentados. A idéia de incorporá-la no sistema de geradores de código que definimos teve origem após o exame de POSTPASS, IPS e RASE presentes em Marion [4].

Os métodos existentes para alocação e escalonamento [11], [20], [5] etc. usam para cada uma destas funções um modelo de grafo para representar o programa fonte. Como o significado dos vértices e arestas nestes grafos são diferentes, é difícil obter uma simples combinação dos mesmos. A estratégia de Pinter provê apenas um modelo de grafo, o *grafo de interferência paralelizável*, para representar o programa fonte para as funções de alocação e escalonamento. Com este modelo é possível usar o algoritmo de coloração também nas arquiteturas superescalares e obter uma alocação de registradores que não restringe a ação do escalonador, pelo contrário, aumenta a eficiência do escalonamento de instruções e determina o nível de comunicação necessária entre estas duas funções. A ênfase é dirigida ao alocador de registradores. Ele é projetado de tal forma que uma ótima utilização do conjunto de registradores pode ser obtida tanto, quando há um número suficiente de registradores disponíveis ou, quando este número é muito pequeno. Situações onde há necessidade de derramamento de alguns valores para a memória aplicam-se ao mesmo tempo neste grafo heurísticas do escalonador e do alocador de registradores. Neste processo novas heurísticas são definidas.

3.1 Apresentação Informal da Estratégia

A existência de dependência falsa limita o potencial de paralelismo implementado pelo escalonador. Seja (u, v) uma aresta de dependência de dados no grafo de escalonamento gerada após a alocação de registradores; a aresta (u, v) é uma aresta de dependência falsa se e somente se u e v podem ser escalonadas juntas de acordo com o grafo de escalonamento para o código quando o mesmo usa pseudo-registradores. Para um dado bloco básico, definem-se o grafo não dirigido de dependências falsas, $G_f = (V_f, A_f)$ e $G_c = (V_c, A_c)$ o grafo de escalonamento antes da alocação de registradores. Em G_c todas as instruções usam pseudo-registradores e todas as restrições de precedência estão presentes. $V_f = V_c$. Após a construção de G_c considere o conjunto de arestas de um fecho transitivo de G_c e defina A_t como sendo este mesmo conjunto de arestas após a eliminação de suas direções. Neste novo grafo acrescentam-se a A_t todas as restrições que não sejam de precedência e que descrevam restrições de arquitetura. O conjunto de arestas A_f compreende os pares (u, v) tais que $u, v \in V_f$, $u \neq v$ e $(u, v) \notin A_t$. Em alguns algoritmos de escalonamento estas arestas não são inseridas no grafo, o próprio algoritmo trata deste caso tentando escalonar uma operação de ponto flutuante seguida de uma operação de ponto fixo, e vice-versa. No modelo apresentado nesta seção todas estas restrições são incluídas explicitamente no grafo. Quanto mais arestas estiverem presentes no grafo melhor será o resultado, porque o modelo usa somente arestas que estão no complemento do grafo construído. Estas arestas refletem o paralelismo existente na máquina para um dado programa. Como um pseudo-registrador nunca é redefinido, o grafo de escalonamento, G_c não possui anti-dependência e dependência de saída, assim o conjunto A_t contém exatamente as mesmas restrições do escalonador.

Após construir os grafos G_f e G_r (grafo de interferência, Seção 1), é possível definir o *grafo de interferência paralelizável*. Com a presença das arestas destes dois grafos, a coloração ótima assegura que não haverá derramamento de código ou introdução de dependência falsa. Observe também, que a união destes dois grafos é o mesmo que modificar o algoritmo de coloração para

funcionar da seguinte forma: u e v não podem ser escalonadas juntas se há uma dependência falsa no grafo. Contudo, note que a solução é introduzindo o teste, transforma Formalmente, define-se o *grafo de interferência* como:

$$\bullet V = V_r$$

$$\bullet A = A_r \cup \{(u, v) \mid t_q(u, v) \in V_r\}$$

Em geral podem existir arestas de interferência em A_r que não aparecem em A_s . Olhe, podem não aparecer em A_r , ou podem não aparecer em A_r uma vez final, apresentado na próxima seção comandos baseados em heurísticas.

Após a construção do *grafo de interferência* com vértices extras, como *stores* e *loads*. Estas partes do grafo estendido. Define-se o *grafo de interferência* como:

$$\bullet V = V_s$$

$$\bullet A = A_r \cup \{(u, v) \mid t_q(u, v) \in V_s\}$$

Neste novo grafo uma aresta é adicionada no mesmo ciclo ou os vértices. Assim, para cada vértice v as arestas inseridas serão escalonadas como no algoritmo (Seção 2.1).

3.2 Gerenciamento do Derramamento

Uma ótima coloração do *grafo de interferência* para arquiteturas superescalares com o objetivo de obter o máximo de paralelismo para o escalonador.

Mas como o problema da coloração é menor que o conjunto mínimo de vértices, são efetuados, armazenando-se em memória. Tendo em vista este mapeamento de registradores tal como no *grafo de interferência* como no escalonamento. Estas

O primeiro tipo de heurística é a remoção de algumas arestas que não são consideradas no intervalo de escalonamento quando, devido

⁸ não sendo considerado no intervalo

Variável e

deficiências dos
o que definimos

para cada uma
o significado dos
ção dos mesmos.
a paralelizável,
ento. Com este
escalares e obter
entrário, aumenta
necessária entre
projetado de tal
tanto, quando há
muito pequeno.
mória aplicam-se
radores. Neste

do pelo escalo-
to gerada após a
samente se u e
ódigo quando o
o não dirigido de
es da alocação
as as restrições
o conjunto de
nto de arestas
as restrições que
de arestas A_f
oritmos de esca-
caso tentando
p, e vice-versa.
mente no grafo.
que o modelo
estas refletem o
istrador nunca
ância de saída,

afinir o grafo de
loração ótima
falsa. Observe
coloração para

funcionar da seguinte forma: use o grafo G , mas ao colorir dois vértices, se ele não contém uma dependência falsa no grafo de interferência original (G_r), não os colora com a mesma cor. Contudo, note que a solução adotada é muito mais elegante, ao invés de modificar o algoritmo introduzindo o teste, transformações são efetuadas nos dados e o algoritmo permanece o mesmo. Formalmente, define-se o *grafo de interferência paralelizável*, $G = (V, A)$, de um bloco básico como:

- $V = V_r$
- $A = A_r \cup \{(u, v) \mid (u, v) \in A_f \text{ e } u, v \in V\}$

Em geral podem existir arestas em A_s que não estão em A_r e vice-versa. Por exemplo, uma aresta de interferência em A_r a qual aparece devido a ordem seqüencial do código de entrada pode não aparecer em A_s . Olhando na direção inversa, restrições baseadas nas arquiteturas podem não aparecer em A_r , ou arestas $(u, v) \in A_s$ relativas ao fluxo de dependência de dados pode não aparecer em A_r uma vez que a mesma corresponde ao último uso de u ⁸. No algoritmo final, apresentado na próxima seção é introduzido um passo que melhora a ordem inicial dos comandos baseados em heurísticas de escalonamento.

Após a construção do *grafo de interferência paralelizável* ele pode ser modificado para incluir vértices extras, como *stores* e arestas a fim de representar mais informações para o escalonador. Estas partes do grafo estendido não fazem parte no algoritmo de alocação de registradores. Define-se o *grafo de interferência paralelizável estendido* para um bloco básico como:

- $V = V_s$
- $A = A_r \cup \{(u, v) \mid (u, v) \in A_f \text{ e } u, v \in V\}$

Neste novo grafo uma aresta entre dois vértices significa que as duas operações podem ser escalonadas no mesmo ciclo ou os dois vértices representam intervalos ativos que não são disjuntos. Assim, para cada vértice v as arestas $(v, u) \in A_f \subseteq A$ provê a lista de instruções aptas com v a serem escalonadas como no algoritmo de escalonamento baseado em listas usado em [10] (Veja Seção 2.1).

3.2 Gerenciamento do Derramamento de Código

Uma ótima coloração do *grafo de interferência paralelizável* induz uma ótima alocação de registradores para arquiteturas superescalares porque ela provê a atribuição de um número mínimo de registradores com o objetivo de não gerar derramamentos de valores em uso e mantém a opção de paralelismo para o escalonador sem dependências falsas no grafo resultante.

Mas como o problema da coloração mínima é NP-completo e, em geral o número de registradores é menor que o conjunto mínimo de cores da melhor coloração obtida. Na prática, derramamentos são efetuados, armazenando-se temporariamente os valores de alguns pseudo-registradores na memória. Tendo em vista estes fatos, o problema da alocação, neste caso, é encontrar um mapeamento de registradores tal que o custo de derramamento seja mínimo. Para isto aplicam-se sobre o *grafo de interferência paralelizável* as mesmas heurísticas usadas tanto na alocação como no escalonamento. Estas heurísticas estão embutidas no método de coloração de Chaitin.

O primeiro tipo de heurística elimina arestas do grafo. A questão de qual aresta deve ser eliminada pode implicar em considerações tanto do escalonador como do alocador. Pode-se considerar a remoção de algumas arestas que evitam dependências falsas, neste caso, está-se fazendo o trabalho do escalonador quando, devido a demanda por registradores, algumas opções de paralelismos

⁸ não sendo considerado no intervalo de pseudo-registradores ativos.

são perdidas. Pode ser considerado também preservar algumas arestas que prometem um bom paralelismo e decidir remover arestas de interferência as quais podem ocasionar derramamentos de código⁹. As seguintes regras são usadas: (1) remova todas as arestas de $A - A_r$ para as quais o escalonamento paralelo de duas instruções possui a menor prioridade. Tal prioridade pode envolver o escalonamento prévio de uma instrução a qual é a última no caminho crítico das arestas de precedência (em A_s). (2) Evite a remoção das arestas em $A_f \cap A_r$. Estas arestas são usadas pelo escalonador e pelo alocador.

Outra abordagem é gerar a função heurística h , usada em alocadores para selecionar a definição que deve ser derramada. Esta função é definida por: $h(v) = \text{cost}(v)/\text{degree}(v)$, $\text{cost}(v)$ representa o custo de manter o valor v em um registrador, e $\text{degree}(v)$ representa o grau de v no grafo.

Sugere-se melhorar esta função atribuindo pesos às arestas do *grafo de interferência paralelizável*. A função de peso $w : E \rightarrow \mathcal{R}$ pode ser gerada usando as regras (1) e (2) abaixo que distinguem os vértices que evitam o derramamento daqueles que preservam o paralelismo: (1) aloque um único registrador para as definições u e v onde $(u, v) \in A - A_r$. Esta regra pode não provocar um derramamento, mas inibe o paralelismo. (2) Aloque 2 registradores para as definições u e v onde $(u, v) \in A_f \cap A_r$, ambos previnem um derramamento e permitem o paralelismo de duas definições. Em geral, é preferível escolher o paralelismo que possivelmente irá se materializar do que o custo de derramar alguns valores para a memória, isto porque o primeiro utiliza uma unidade livre comparado com o custo do derramamento. Cada aresta deste tipo possui um preço que depende da máquina dada. Em particular, se todas as arestas em $A - A_r$ possuem peso 1, então a função h tem o efeito tradicional.

Seja $\text{in}(v)$ o conjunto dos vértices onde cada um compartilha uma aresta com v . A nova função h pode ser: $h^*(v) = \frac{\text{cost}(v)}{\sum_{u \in \text{in}(v)} w(\{u, v\})}$. A função cost , em geral, é uma função do nível de aninhamento da instrução e não pode ser mudada.

O *grafo de interferência paralelizável* pode incorporar informações de dependência do correspondente grafo de escalonamento da seguinte forma: estende-se primeiramente o *grafo de interferência paralelizável* $G = (V, A)$ adicionando a todo vértice $v \in V$ um valor $EP(v)$ representando o menor tempo para escalonar v . Os valores EP são calculados a partir do grafo de escalonamento (G_s). Durante este estágio, o número de atrasos associados às arestas do grafo de escalonamento podem ser usados para gerar valores mais precisos para EP permitindo que o *grafo de interferência paralelizável* pode ser usado tanto para alocação de registradores como para o posterior escalonamento de instruções.

3.3 Pares de Registradores

O método de coloração de grafos de Chaitin, bem como o algoritmo proposto por Pinter [17] não discutem o problema de pares de registradores. Em seus algoritmos, um pseudo-registrador v pode ser removido do grafo, durante a fase de simplificação se for garantido existir um registrador físico para ele durante a fase de coloração. Isto significa que o seu grau, dado pela função $\text{in}(v)$, deve ser inferior ao número de registradores disponíveis.

A necessidade por mais de um registrador para um dado vértice v muda a sua definição de

⁹O derramamento pode ser evitado somente quando se atribuem para dois vértices deste tipo cores diferentes apesar da ausência da aresta de interferência.

grau. Um vértice v é considerado de registradores alocáveis [5]. C que um vértice v pode ser remo simplificação se o predicado is-s onde: $\text{degree}(v) = \sum_{q \in \text{in}(v)} \text{need}$ r representa os registrad $\text{need}(v)$ representa o nú

Com esta modificação introduzi *paralelizável* mesmo que a dem pares de registradores não possi nesta seção, que é manter o pa *de interferência paralelizável*, el outro esquema. A reutilização de pode gerar uma coloração de complementando, propõe-se aqui se derramamento for considerado como derramado, então nenhum pode ser considerado na escolha registradores pode dificultar m $h^*(v)$. Assim, o tamanho passa a

O algoritmo para alocação de regi de escalonamento e o método de pseudo-registradores, primeiram se o mapeamento $EP : V \rightarrow N$ percorra o grafo G_s , processando nores. Sempre que operações com limitações de arquitetura, seleção EP de cada vértice do conjunto o vértice, em G_s . Quando este p tente com a ordem parcial dos n segunda parte, o método de color *lizável* $G = (V, A)$ do trecho de p em análise e inicialize a lista de

```

G' := G
; Fase de Simplificação
while G não está vazio do
    while existe vértice v ∈ V tal
        while existe vértice v ∈ V tal
            interferencia são consi
                use consideraçõe
                dependência fa
                escalonamento d
                alguns possível
            while existe um
        end do
    if V não está vazio then { E
end do

```


Um vértice v é considerado sem restrições se $degree(v) + need(v)$ é menor que o número de registradores alocáveis [5]. Generalizando estas idéias propomos que é possível determinar que um vértice v pode ser removido do *grafo de interferência paralelizável* durante a fase de simplificação se o predicado $is-safe(v)$ é verdadeiro, sendo: $is-safe(v) = degree(v) + need(v) \leq r$ onde: $degree(v) = \sum_{q \in in(v)} need(q)$.

r representa os registradores físicos disponíveis.

$need(v)$ representa o número de registradores físicos necessários à v .

Com esta modificação introduzida é possível obter uma coloração do *grafo de interferência paralelizável* mesmo que a demanda dos vizinhos de um vértice v seja maior que r . Como pares de registradores não possuem nenhuma relação com o objetivo do algoritmo proposto nesta seção, que é manter o paralelismo de instruções evitando dependências falsas no *grafo de interferência paralelizável*, eles podem ser incorporados neste algoritmo como em qualquer outro esquema. A reutilização de cores em vizinhos cujas arestas não se interferem neste grafo pode gerar uma coloração de G sem alterar o objetivo do algoritmo proposto. Além disto, complementando, propõe-se aqui, como sugere [19] que a fase de coloração seja tentada mesmo se derramamento for considerado necessário. Se esta fase conseguir colorir um vértice v marcado como derramado, então nenhum derramamento é gerado para v . Observe também que o tamanho pode ser considerado na escolha do candidato a derramar. Como um candidato que usa mais registradores pode dificultar mais o processo de coloração, ele deve ser penalizado na função $h^*(v)$. Assim, o tamanho passa a ser mais um divisor de custo.

O algoritmo para alocação de registradores é composto de duas partes, a aplicação das heurísticas de escalonamento e o método de coloração. Dado um trecho de programa representado por pseudo-registradores, primeiramente gera-se o grafo de escalonamento, G_s . Em seguida, calcula-se o mapeamento $EP: V \rightarrow N$ para o escalonamento usando suas heurísticas da seguinte forma: percorra o grafo G_s processando os vértices incrementando os valores EP , primeiramente os menores. Sempre que operações com o mesmo EP não puderem ser escalonadas juntas devido a limitações de arquitetura, selecione estas operações para serem retardadas; incremente o valor EP de cada vértice do conjunto adiado e atualize os valores EP de todos os caminhos deixando o vértice, em G_s . Quando este processo terminar, selecione uma ordem linear que seja consistente com a ordem parcial dos novos valores EP e reordene o trecho do programa de acordo. A segunda parte, o método de coloração tem início com a geração do *grafo de interferência paralelizável* $G = (V, A)$ do trecho de programa. Atribua a r o número de registradores da arquitetura em análise e inicialize a lista de derramamento para ser vazia.

$G' := G$

; Fase de Simplificação

while G não está vazio do

while existe vértice $v \in V$ tal que $is-safe(v)$ do Remova v ; Atualiza G end do

while existe vértice $v \in V$ tal que $is-safe(v)$ quando somente arestas de interferência são consideradas do

use considerações sobre o escalonamento para remover de G' e G uma aresta de dependência falsa que não está em A_r , ou seja, uma aresta (v, u) para a qual o escalonamento de u com v provê uma contribuição mínima, abandonando assim alguns possíveis paralelismos.

while existe um vértice $v \in V$ tal que $is-safe(v)$ do Remova v ; Atualize G end do

end do

if V não está vazio then { Escolha $v \in V$ com o mínimo $h^*(v) = \frac{cost(v)}{\sum_{u \in in(v)} w(\{u, v\})}$
Remova v e atualize G ; Coloque v na lista de derramamento }

end do

; Fase de Seleção de Cor

if lista de derramamento está vazia then

Colora os vértices em ordem reversa da remoção, isto é feito reconstruindo G' vértice a vértice.

else { ; Fase de Derramamento

Coloque cada v na lista de derramamento; Repete o procedimento de coloração. }

3.4 Comparação com RASE

A idéia inicial era inserir a estratégia proposta em Marion [5] baseado no esquema modular da parte referente à estratégia de geração de código deste sistema. Esta inserção possibilitaria obter uma avaliação quantitativa de seu desempenho em relação às estratégias existentes naquele sistema, antes de incorporá-la ao gerador de código que projetamos. Contudo, considerando que o modelo apresentado consiste em um coerente conjunto de heurísticas para reduzir o volume de código derramado e que a avaliação teórica destas heurísticas é difícil porque, para os algoritmos de coloração e derramamento, não há como comparar quantitativamente o desempenho entre duas funções heurísticas para todos os programas. Optamos em uma primeira etapa, por uma avaliação qualitativa do modelo proposto em relação às estratégias de Marion, deixando a avaliação quantitativa para o futuro.

A alocação de registradores de Marion não é satisfatória. Bradlee não se preocupa com a introdução de dependências falsas. Como a ordem na qual pseudo-registradores são coloridos é baseada em prioridades e não proximidade, pseudo-registradores na mesma vizinhança podem ser atribuídos ao mesmo registrador físico. Como Marion faz o escalonamento de instruções em paralelo com a alocação, derramamentos ocorrem com mais frequência. Além disto, a forma utilizada para alocar registradores aos *pseudo-registradores* locais acarreta em dois problemas que complicam o algoritmo de escalonamento e muitas vezes provoca o derramamento desnecessariamente. O primeiro problema diz respeito ao esforço necessário para evitar recarregar um registrador derramado quando ele não pode ser usado. A solução adotada pelo escalonador para este problema é marcar cada recarga e escaloná-la somente se alguns sucessores só tenham predecessores de recarga. Isto evita recarregar muito cedo, mas às vezes força a ocorrência de recarga muito tarde.

O segundo problema está relacionado com a ativação de funções. Durante o pré-escalonamento, o escalonador anota quantos *pseudo-registradores* locais estão ativos através de cada chamada no bloco básico. O alocador usa esta informação na tentativa de fornecer ao bloco básico um número suficiente de registradores para suprir suas necessidades. Contudo, até que uma chamada de função seja escalonada, o escalonador não tem conhecimento de quais *pseudo-registradores* estarão ativos através da chamada. Uma alternativa para solucionar este problema é atrasar a atribuição de registradores até depois do escalonamento. Como o escalonador permanece dentro do limite de registradores pré-fixado, derramamento pode não ser necessário, mas algumas movimentações entre registradores podem ser necessárias devido a necessidades de registradores específicos para passagem de parâmetros e pares de registradores.

Uma solução mais adequada para os dois problemas mostrados é permitir que o escalonador exceda ao limite de registradores, e então, efetuar alocação local depois do escalonamento. Naturalmente, com esta solução, derramamentos terão que ser inseridos, tornando necessário, também, o escalonamento dos mesmos. Contudo isto simplifica o algoritmo de escalonamento e pode melhorar o desempenho do tempo de compilação e o escalonamento produzido.

No modelo apresentado nesta seção, durante a construção do grafo de interferência, o ponto

final do intervalo de uma variável e o reuso do registrador no mesmo

Algumas estratégias, por exemplo, registradores locais que podem ser de estabelecer um limite para as possibilidades de explorar o paralelismo mostrada nesta seção.

Concluindo, os pontos positivos resumidos nos seguintes itens:

- (1) o escalonador não é afetado;
- (2) O grafo de escalonamento contém menos vértices;
- (3) O grafo para alocar registradores é mais simples;
- (4) O grafo de interferência para o escalonamento e alocação;
- (5) O algoritmo de escalonamento usar o algoritmo proposto são simples, contudo capazes

4 Conclusões

Neste trabalho investigamos um problema da interdependência de instruções. Apresentamos três abordagens onde cada uma delas representa uma estratégia de escalonamento. Resultados da análise de custo de compilação de estratégias não são realmente necessárias. Entre estas duas funções é necessária a estratégia RASE, alguns pontos estão relacionados ao seu custo. O algoritmo deve ser executado para cada conjunto de instruções. As decisões são complicadas.

Tentando resolver o problema da permanência sem uma solução simples para a grande parte do problema. Esta alocação de registradores é efetuada sem dependências falsas no grafo de escalonamento e a estrutura de dados mais simples para coloração de grafos e paralelismo [10] é possível efetuar a alocação sem perda do paralelismo inerente do grafo final de escalonamento.

A principal contribuição deste trabalho para escalonamento de instruções

final do intervalo de uma variável viva não é considerado como parte do intervalo. Isto permite o reuso do registrador no mesmo comando de seu último uso.

Algumas estratégias, por exemplo, IPS [11] e RASE [3] estabelecem um número de pseudo-registradores locais que podem estar simultaneamente ativos em cada bloco básico. O fato de estabelecer um limite para o número de pseudo-registradores locais resulta na perda de possibilidades de explorar o paralelismo entre instruções. Este fato não acontece na estratégia mostrada nesta seção.

Concluindo, os pontos positivos desta abordagem em relação a abordagem de Marion podem ser resumidos nos seguintes itens:

- (1) o escalonador não é afetado pelo alocador de registradores.
- (2) O grafo de escalonamento continua sendo G_c .
- (3) O grafo para alocar registradores é o *grafo de interferência paralelizável*.
- (4) O *grafo de interferência paralelizável estendido* pode ser usado para as duas funções de escalonamento e alocação.
- (5) O algoritmo de escalonamento não precisa ser complicado como RASE, podendo inclusive usar o algoritmo proposto por Gibbon et al [10] cujas técnicas heurísticas empregadas são simples, contudo capazes de prover um bom escalonamento.

4 Conclusões

Neste trabalho investigamos um dos problemas mais sérios para as arquiteturas superescalares, o problema da interdependência entre as funções de alocação de registradores e escalonamento de instruções. Apresentamos três estratégias de geração de código, RASE, IPS e POSTPASS [3], onde cada uma delas representa um grau diferente de acoplamento entre as funções de alocação e escalonamento. Resultados da análise destes modelos demonstraram que implementações e custo de compilação de estratégias que acoplam fortemente estas duas funções, como RASE, não são realmente necessárias. Contudo, pelo menos um nível intermediário de comunicação entre estas duas funções é necessário para produzir um bom código. Vimos vários problemas com a estratégia RASE, alguns deles sugerimos modificações. Mas o problema maior de RASE está relacionado ao seu custo. O escalonador de instruções além de ser muito mais complexo deve ser executado para cada bloco básico de três a quatro vezes, dependendo do número de conjuntos de instruções. As decisões sobre derramamento durante o escalonamento também são complicadas.

Tentando resolver o problema da interdependência entre o alocador e o escalonador que ainda permanecia sem uma solução satisfatória, foi descrita uma estratégia na Seção 3 que resolve grande parte do problema. Esta estratégia é baseada no algoritmo de Pinter [17]. Nele, a alocação de registradores é efetuada antes do escalonamento de instruções e não introduz dependências falsas no grafo de escalonamento. A abordagem descrita demonstrou que através de uma estrutura de dados mais sofisticada e usando algoritmos bem estabelecidos na literatura para coloração de grafos e para escalonamento, como por exemplo aqueles propostos por [7] e [10] é possível efetuar a alocação de registradores antes do escalonamento de instruções sem perda do paralelismo inerente da máquina em análise e sem introduzir dependências falsas no grafo final de escalonamento.

A principal contribuição deste trabalho compreende a avaliação e melhoramento do algoritmo para escalonamento de instruções combinado com alocação de registradores. Os principais re-

sultados obtidos foram: (1) A identificação do nível de interdependência satisfatório entre o escalonamento de instruções e a alocação de registradores. (2) A proposta da modificação necessária ao método de alocação de registradores de Pinter com o objetivo de tratar o uso de mais de um registrador por variável (Seção 3.3). A continuação do trabalho compreende a implementação e avaliação do algoritmo de alocação de registradores proposto e a implementação das modificações propostas para RASE e sua futura incorporação em GGC0 [?] como uma segunda opção para escalonamento.

Referências

- [1] D. Bernstein, M.C. Golumbic, R.Y. Pinter, D.Q. Goldin, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *ACM Sigplan Conference on Programming Language Design and Implementation*, July 1989. ACM Sigplan Notices, 24(7), July 1989.
- [2] David Bernstein and Izidor Gertner. Scheduling expression on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1), January 1989.
- [3] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The effect on risc performance of register set size and structure versus code generation strategy. In *International Symposium on Computer Architecture*, May 1991.
- [4] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for riscs. In *ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991.
- [5] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The marion system for retargetable instruction scheduling. In *ACM Sigplan Conference on Programming Language Design and Implementation*, July 1991. ACM Sigplan Notices 26(7).
- [6] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *ACM Sigplan Notices*, 10(7), July 1989. ACM Sigplan Conference on Programming Language Design and Implementation, 1989.
- [7] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *ACM Sigplan Notices*, 17(6), June 1982. ACM Sigplan Symposium on Compiler Construction, 1982.
- [8] Joseph A. Fisher. Trace scheduling: A technique for global microcode compactation. *IEEE Transactions on Computers*, 30(7), July 1981.
- [9] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dump machine. *ACM Sigplan Notices*, 19(6), June 1984. Proceedings of the ACM Sigplan, Symposium on Compiler Construction, 1984.
- [10] Phillip B. Gibbons, and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction*, July 1986. ACM Sigplan Notices 21(7).
- [11] James R. Goodman and Wei-Chung-Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing - Conference ACM-PRESS Proceedings*, St. Malo France, July 1988.
- [12] _____.
- [13] Monica Lam. Software pipelining. *Proceedings of the Sigplan'88*, Atlanta Georgia, June 1988.
- [14] D. Landskov et al. Local microcode. March 1980.
- [15] Auslander Marc and Hopkin. *Symposium on Compiler Construction*. Notices 17(6).
- [16] Krishna V. Palem and Barbara G. Ryder. Register allocation for risc machines. In *ACM Symposium on Compiler Construction*, June 1993.
- [17] Shlomit S. Pinter. Register allocation. *Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction*, Albuquerque, New Mexico, June 1986.
- [18] Charles N. Proebsting, Todd A. Anderson, and David A. Duff. Delayed-lead architectures. In *Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction*, July 1986. ACM Sigplan Notices 26(7), July 1986.
- [19] Galileu B. Souza. Técnicas de otimização de código. Tese de doutorado, Departamento de Ciência da Computação, Universidade Federal do Rio de Janeiro, 1991.
- [20] H. S. Warren Jr. Instruction scheduling. *Journal of Research and Development*, 1982.

- [12] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3), July 1983.
- [13] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the Sigplan'88 Conference on Programming Language Design and Implementation*, Atlanta Georgia, June 1988.
- [14] D. Landskov et al. Local microcode compaction techniques. *ACM Computing Surveys*, 12, March 1980.
- [15] Auslander Marc and Hopkins Martin. An overview of the pl.8 compiler. In *ACM Sigplan Symposium on Compiler Construction*, Boston Massachusetts, June 1982. ACM Sigplan Notices 17(6).
- [16] Krishna V. Palem and Barbara B. Simons. Scheduling time-critical instructions on risc machines. In *ACM Symposium on Principles of Programming Languages*, January 1990.
- [17] Shlomit S. Pinter. Register allocation with instruction scheduling: a New Approach. In *Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation*, Albuquerque NM, June 1993. ACM Sigplan Notices - Vol. 28 number 6 June-1993.
- [18] Charles N. Proebsting, Todd A. and Fischer. Linear-time, optimal code scheduling for delayed-lead architectures. In *Proceedings of the ACM Sigplan'91 Conference on Programming Language Design and Implementation*, Toronto Ontario Canada, June 1991. ACM Sigplan Notices 26(7), July 1991.
- [19] Galileu B. Souza. Técnicas de otimização de código para arquiteturas RISC. Master's thesis, Departamento de Ciência da Computação, UNICAMP Campinas, SP, Maio 1992.
- [20] H. S. Warren Jr. Instruction scheduling for the ibm risc system/6000 processor. *IBM Journal of Research and Development*, 34(1), January 1990.

