PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, No. M11/94

Editor: Prof. Carlos José Pereira de Lucena

Sistema Gerador de Geradores de Código para Arquiteturas Superescalares

Mariza Andrade da Silva Bigonha

José Lucas Mourão Rangel Netto

Agosto/1994

Para obter cópias:
Rosane T. L. Castilho
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22.453 - Rio de Janeiro, RJ.
Brasil

Abstract

Modern computer architectures have motivated research for more efficient compiler techniques. These new architectures, however, delegate the solution of the most complicated problems in code generation to the compiler. The focus of this paper is the design of a retargetable code generator system for superscalar architectures. We also show the results of the study of several problems related to code generation to these processors.

Keywords: Superscalar Architecture, Register Allocation, Instruction Scheduling, Code Generation, Optimization.

Resumo

Arquiteturas mais modernas de computadores motivam pesquisas por técnicas de implementação de compiladores mais eficientes. Isto, contudo, acarreta maior complexidade dos compiladores porque estes novos computadores alcançam a eficiência, delegando-lhes a solução de problemas complexos de geração e otimização de código. Mostramos neste trabalho o projeto de um sistema gerador de geradores de código apropriado para arquiteturas superescalares. Mostramos também resultados de um estudo sobre vários problemas relacionados com a geração de código para estes processadores.

Palavras-chave: Arquitetura Superescalar, Alocação de Registradores, Escalonamento de Instruções, Geração de Código, Otimização.

Sumário

1 Introdução **1**

2 Metodologia de Construção de Geradores de Código **2**

3 O Sistema GGCO **3**

3.1 O *Front-end* 3

3.2 Gerador de Código 4

3.2.1 Seletor de Código 5

3.2.2 Escalonamento de Instruções 6

3.2.3 Alocação de Registradores 7

4 A LDA **8**

4.1 Mapeamento com a Linguagem Intermediária 10

5 Filosofia da Definição Formal de LDA **11**

5.1 Principais Tabelas Geradas 12

6 Modelos de Pilha de GGCO **13**

7 Conclusões **15**

1 Introdução

O alvo do sistema gerador de geradores de código descrito neste trabalho são os processadores superescalares. Estes processadores são uma evolução das arquiteturas `RISC` (*Reduced Instruction Set*). As arquiteturas superescalares possuem várias características em comum. As principais são a habilidade de executar mais de uma instrução por ciclo e a incorporação de várias unidades funcionais que podem operar em paralelo. A maior vantagem desta última característica é a habilidade de explorar o paralelismo em nível de instrução, pela execução simultânea de instruções em unidades individuais. A forma como esta operação concorrente é extraída de um conjunto de instruções seqüencial varia de máquina para máquina. Contudo, independente do mecanismo usado para buscar e executar instruções, o compilador deve administrar de forma eficiente a alocação de registradores e o escalonamento de instruções.

A alocação consiste no mapeamento das variáveis declaradas no programa fonte e das variáveis temporárias, geradas durante o processo de compilação, em registradores da máquina. Ela é considerada ótima se as variáveis permanecerem nos registradores todo seu ciclo de vida. Escalonamento é o processo de decidir a ordem das instruções de forma que elas possam ser executadas em unidades funcionais distintas com o objetivo de minimizar o tempo total de processamento do programa. Dois pontos são fundamentais nesta movimentação de instruções: boa utilização da máquina em análise e preservação da semântica do programa.

O maior problema, que ainda permanecia sem uma solução satisfatória até esta data, está relacionado ao nível de comunicação entre a alocação e escalonamento. Questiona-se até que ponto estas duas funções devem interagir para se produzir um bom código e se o escalonamento de instruções ineficiente está associado à falta de comunicação entre eles. Trabalhos nesta linha [Bradlee et al., 1991a, Benitez and Davidson, 1988, Kerns, 1993] e [Bigonha, 1994a] sugerem algumas soluções. As linguagens de descrição de arquiteturas caracterizam outro problema que até hoje ainda não tinha uma solução satisfatória [Bigonha and Rangel, 1994].

O objetivo deste trabalho é apresentar o projeto de uma ferramenta de auxílio à implementação de compiladores de boa qualidade, o GGCO (Gerador de Geradores Redirecionáveis de Código Otimizado). Fazem parte deste sistema: (1) o projeto e definição formal da sintaxe e semântica de uma linguagem para descrever o conhecimento embutido nas arquiteturas de computadores (LDA), tendo em vista arquiteturas superescalares; e (2) a identificação do nível de interdependência satisfatória entre o escalonamento e a alocação.

2 Metodologia de Construção de Geradores de Código

Os geradores de código desenvolvidos na década passada deram mais ênfase à seleção de instruções para as arquiteturas CISC (*Complex Instruction Set Computers*), por exemplo, o sistema PO de Davidson [Davidson, 1981] e sucessores, o sistema CODEGEN de R. Henry [Henry, 1987], o sistema AutoCode de Costa [Costa, 1990]. Como estas arquiteturas implementavam as operações mais comuns de várias formas, escondendo detalhes de *pipelining* e outras características, geradores de código para estas arquiteturas se baseavam em especificações de máquina que permitiam, na maioria das vezes, que instruções fossem selecionadas utilizando-se de casamento de padrões. Estes geradores não usavam técnicas de escalonamento e, muitas vezes, não efetuavam alocação global de registradores. Em contrapartida, as RISCs implementam a maior parte das operações de uma única forma e expõem ao gerador e otimizador a estrutura da *pipeline* e custos das unidades funcionais.

Devido à mudança de ênfase, problemas relacionados à geração de código para arquiteturas RISC são diferentes daqueles para arquiteturas CISC. Em primeiro lugar, para um gerador de código RISC redirecionável[1], a especificação de máquina deve capturar informações de escalonamento, incluindo latência das operações e conflitos de recursos. Em segundo lugar, como a seleção de instruções nas arquiteturas RISC é relativamente simples, a interação entre a alocação e a seleção de instruções é considerada menos importante. Por outro lado, a interação entre a alocação e o escalonamento é significativa e o escalonador necessita registradores para sobrepor a execução de operações independentes. Redirecionamento dos geradores continua importante para estas novas arquiteturas. Primeiro porque o desempenho de novos processadores continua sendo aprimorado, induzindo os usuários a melhorarem seus sistemas. Segundo porque o volume de *software* que deve ser transportado para uma nova máquina é grande o suficiente para proibir o uso intensivo de código em linguagem de montagem. Assim, para que uma nova arquitetura seja competitiva no mercado, seu compilador deve ser capaz de gerar código de boa qualidade. Terceiro porque podem haver diferenças acentuadas entre diferentes famílias de processadores: Intel i860 e MC88100 [Intel, 1989, Motorola, 1990]; ou mesmo diferenças significativas entre processadores de uma mesma família: MC68000 e MC88100 [Motorola, 1984, Motorola, 1990]. Sistemas redirecionáveis tornam possível a tarefa de construir geradores de boa qualidade em um período de tempo relativamente curto.

Praticamente não existem sistemas de geração de código redirecionável especificamente projetado para arquiteturas RISC. Muito menos ainda para arquiteturas superescalares. Os sistemas Gnu [Stallman, 1989] e Marion [Bradlee et al., 1991b] são os únicos conhecidos. Atualmente, existem pelo menos duas versões do sistema Gnu que já incorporam um escalonador. Uma delas usa o algoritmo de Gibbon et al. [Gibbons et al., 1986], a alocação

[1] Um gerador de código redirecionável é aquele que pode ser redefinido automaticamente para gerar código para novas máquinas, a partir de descrições de arquiteturas.

é feita antes do escalonamento e não existe nenhum tipo de comunicação entre eles. A outra versão usa um escalonador desenvolvido por Tiemann [Tiemann, 1989] onde latências dependentes da arquitetura e informações sobre recursos computacionais são encapsulados. Até esta data Marion [Bradlee et al., 1991b] é o único sistema que possui uma linguagem de descrição de máquina, contudo suas primitivas descrevem apenas os componentes básicos das RISCs. Ela não consegue modelar detalhes mais complicados de algumas arquiteturas superescalares, como por exemplo, o sistema de janelas da SPARC, o sistema de prioridade do barramento destino do processador MC88000.

3 O Sistema GGCO

O sistema GGCO recebe como entrada a especificação de um processador em uma linguagem de descrição de arquitetura, LDA (veja Seção 4) e gera uma série de tabelas e funções que representam o resultado do processamento das informações extraídas das principais diretivas de LDA. A arquitetura de GGCO, ilustrada na Figura 1, compreende as seguintes partes: (1) MD.c é o arquivo contendo as tabelas e funções geradas a partir da especificação da arquitetura (veja Seção 5.1); (2) o módulo gen-mdc define a semântica de uma descrição em LDA (veja Seção 5); (3) MD.h é um arquivo predefinido contendo definições das principais estruturas de dados e tipos usados em MD.c; (4) o *front-end* utilizado em GGCO é o programa lcc de Fraser [Fraser and Hanson, 1992]; (5) o *back-end* utilizado é o programa de Bradlee [Bradlee et al., 1991b] e engloba as estratégias de escalonamento e alocação (veja Seção 3.2). Os arquivos MD.c, MD.h, o *front-end* e o *back-end* mostrados na Figura 1 são processados pelo programa Make e produzem o compilador mcc para a arquitetura em questão. Este compilador recebe como entrada um programa fonte em linguagem C, o transforma para uma linguagem intermediária e então passa o controle para o gerador de código que produz um programa objeto semanticamente equivalente ao programa fonte.

3.1 O *Front-end*

Com a interface do programa lcc o módulo *front-end* independente de arquitetura e o módulo do gerador de código dependente de arquitetura estão bem acoplados. A vantagem desta abordagem é que a mesma produz compiladores eficientes e compactos. A desvantagem, como é sabida, complica a manutenção do compilador, consertos no *front-end* afetam o *back-end*. Contudo, como a linguagem para a qual pretende-se gerar código é o *ANSI C* e o foco principal deste trabalho é a questão da geração de código certamente existirão poucas mudanças na linguagem, então esta complicação não é tão importante.

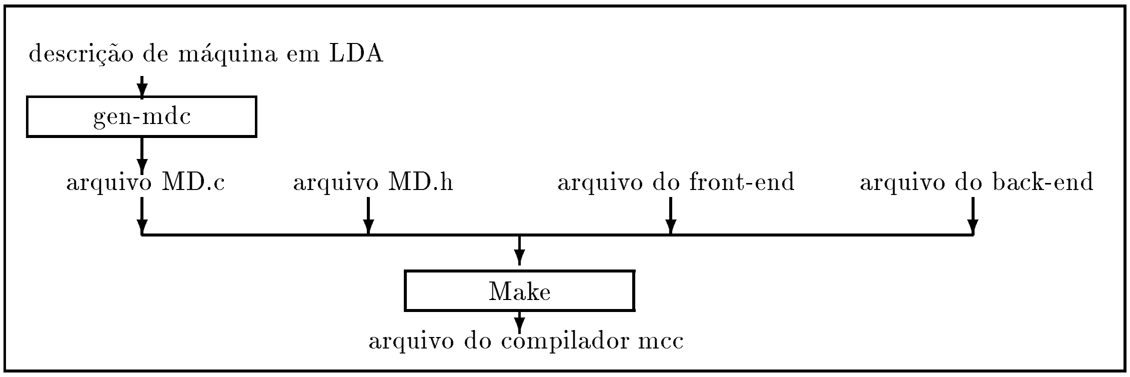

Figura 1: Estrutura do Gerador de Gerador de Código Otimizado-GGCO

O *front-end* do compilador emite código para uma linguagem DAG. Esta linguagem corresponde a linguagem intermediária gerada a partir do programa fonte e provê a configuração inicial para o DAG de código apresentado na Seção 3.2.2. Os vértices de um DAG representam os possíveis operadores do conjunto de instruções de uma arquitetura. O *front-end* transforma todos os operadores de fluxo de controle, "for" , "while" e "if" , etc. em operações de mais baixo nível, como comparação e desvio. Os operadores da linguagem C com efeitos colaterais como: operadores de atribuição, operadores condicionais, etc., são transformados em explícitas operações aritméticas, lógicas, de atribuição e desvios. Dois conjuntos de transformações são aplicados na linguagem intermediária antes de efetuar a seleção de código. O primeiro conjunto efetua alocação de dados usando algumas informações dependentes de máquina. Para cada objeto declarado, o alocador de dados aloca: (1) um pseudo-registrador[2] se o objeto é uma variável automática que pode ser atribuída a um registrador; (2) uma posição na pilha do sistema se o objeto é uma variável automática, mas não pode ser atribuída a um registrador; (3) uma posição na área de dados global, se o modelo de máquina possui tal área e o objeto é estático; (4) um endereço relocável. No segundo conjunto, os endereço dos operadores são expandidos ou substituídos por referências a pseudo-registradores e vértices são inseridos para retornar valores em registradores ou posições da pilha, etc.

3.2 Gerador de Código

Cada gerador de código é construído a partir da descrição de uma determinada arquitetura e efetua a seleção de código, o escalonamento de instruções e a alocação de registradores. GGCO efetua a seleção antes de passar o controle para a estratégia de geração de código.

[2]Pseudo-registradores são registradores criados para conter valores intermediários de expressões e não existe entre eles e o *hardware* dependência alguma.

A composição da estratégia de geração de código de GGCO é a mesma do sistema Marion [Bradlee et al., 1991b]. Ela é composta de duas partes: (1) parte independente da estratégia utilizada e (2) parte dependente. A parte independente da estratégia compreende três componentes lógicos: o construtor do DAG de código (veja Seção 3.2.2), o alocador de registradores e o suporte para o escalonamento. O construtor do DAG de código é responsável pela construção do DAG das instruções de máquina para cada bloco básico[3]. O alocador determina como é utilizado o conjunto de registradores físicos. O suporte para escalonamento é responsável pela verificação de conflitos de recursos e pela manutenção da lista de instruções aptas a serem escalonadas sem provocar atrasos. A parte dependente da estratégia compreende o algoritmo de escalonamento, as tabelas e rotinas geradas a partir da descrição da máquina. Sua estrutura modular permite a substituição ou inserção de novas estratégias de escalonamento e alocação no módulo de geração de código.

3.2.1 Seletor de Código

O seletor de código utiliza o método de seleção descendente recursivo por reconhecimento de padrão para casar vértices das árvores objetos[4] com vértices das árvores padrões[5]. Para cada instrução da máquina existe um padrão e um símbolo de substituição que a identifica. O algoritmo básico da seleção de código examina os padrões em uma dada ordem, selecionando o primeiro que casa com operadores da árvore objeto, e então tenta casar as sub-árvores, começando pela sub-árvore à esquerda. Se for bem sucedido, junta-se o padrão ao vértice do DAG da linguagem intermediária. Se não for bem sucedido o seletor recua e examina o próximo padrão da árvore de padrão. Após a seleção, caminha-se no DAG da linguagem intermediária da esquerda para a direita, *bottom-up*, gerando código de acordo com os padrões anexados aos vértices do DAG. Como a entrada para o seletor de código é um DAG da linguagem intermediária, para acomodá-lo em uma árvore padrão todos os vértices da linguagem intermediária que possuem mais de um pai são forçados em registradores, a menos que sejam constantes que possam ser substituídas por modos de endereçamentos ou operandos imediatos.

[3] Um bloco básico é uma seqüência de instruções na qual só existe uma entrada e uma saída, isto é, não existem instruções de desvios para dentro do bloco ou para fora, exceto a última instrução que pode ser de desvio.

[4] Árvores objeto são sub-árvores do DAG da linguagem intermediária, que constituem a entrada para o seletor de código.

[5] Árvores Padrões são árvores compostas de operadores da linguagem intermediária, derivadas a partir da descrição da máquina pelo gerador de gerador de código.

3.2.2 Escalonamento de Instruções

A estrutura de dados mais importante no processo de escalonamento é o grafo de escalonamento, o DAG de código. Representa-se nesta estrutura de dados os blocos básicos em que foi dividido o programa. Como o escalonamento considerado neste trabalho é feito para blocos básicos, as restrições de precedências consideradas são: restrições baseadas em dependência de dados e dependências de recursos. Dependência de controle existe somente entre blocos básicos e suas correspondentes arestas são derivadas do grafo de fluxo do programa. As dependências de dados entre instruções podem ser verdadeiras ou falsas. As dependências falsas podem ser classificadas em anti-dependência e dependência de saída. Uma dependência de dados de uma instrução u para uma outra v existe se uma das seguintes afirmativas for verdadeira: (1) dependência verdadeira ou de fluxo de dados: registrador definido em u é usado em v; (2) anti-dependência: registrador usado em u é redefinido em v e destrói o valor usado em u; (3) dependência de saída: registrador definido em u é redefinido em v, destruindo o valor definido anteriormente em u.

A abordagem mais utilizada para escalonamento de instruções, denominada *lista de escalonamento* [Hennessy and Gross, 1983], [Fisher et al., 1984], [Gibbons et al., 1986], [Bradlee et al., 1991b], [Warren Jr, 1990], [Goodman and Wei-Chung-Hsu, 1988] funciona da seguinte forma: dado um DAG de código, 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 DAG.

Em uma lista de escalonamento, normalmente 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* e é definida como o comprimento do caminho[6] mais longo através do DAG de código, partindo do vértice da instrução até o vértice folha. Muitas vezes, a *distância máxima* é também denominada *peso* do vértice dentro do DAG. 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. Um segundo nível de heurística 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

[6] O comprimento de um caminho é a soma de todos os valores de rótulos das arestas ao longo do caminho.

o período de latência com o escalonamento de outras instruções. Toda a estratégia de geração de código de GGCO utiliza algoritmos de listas de escalonamento com a distância máxima como primeira heurística.

3.2.3 Alocação de Registradores

Até recentemente, a coloração do grafo de interferência era a abordagem mais usada para resolver o problema da alocação eficiente de valores a registradores físicos em um compilador. Neste grafo, os vértices, representando pseudo-registradores, e as arestas, representando conflitos entre pseudo-registradores em um intervalo de tempo, eram coloridos por um certo número de cores, derramando para a memória alguns valores quando necessário. Com o advento dos processadores superescalares permitindo o paralelismo entre instruções, algoritmos de coloração de grafos considerados ótimos outrora já não se correlacionam com uma boa utilização dos recursos destas máquinas. Para explorar o paralelismo entre instruções, as mesmas devem ser reordenadas. Quando a alocação é feita antes da reordenação, a seleção de registradores pode limitar as possibilidades de escalonamento devido a introdução de dependências falsas com o reuso dos registradores. Quando o escalonamento é feito antes da alocação o número de pseudo-registradores ativos aumenta dando origem a sérias implicações: (1) seu tempo de vida cresce; (2) são necessários mais registradores físicos; (3) ocorrem mais derramamentos para a memória.

Os métodos existentes usam para as funções de escalonamento e alocação diferentes modelos de grafos para representar o programa fonte, [Goodman and Wei-Chung-Hsu, 1988], [Warren Jr, 1990], [Bradlee et al., 1991b] etc. Como o significado dos vértices e arestas nestes grafos são distintas, é difícil obter uma simples combinação dos mesmos. A estratégia proposta para GGCO é diferente destes métodos citados, ela provê apenas um modelo de grafo, o *grafo de interferência paralelizável*, para representar o programa fonte para estas duas funções [Pinter, 1993]. Nesta abordagem, a ênfase é dirigida ao alocador de registradores e o método utilizado para alocação é baseado no trabalho de Chaitin [Chaitin, 1982]. O algoritmo funciona da seguinte forma. Para gerar o *grafo de interferência paralelizável* primeiramente inclui todas as dependências e restrições de recursos explicitamente no grafo de escalonamento. Quanto mais arestas estiverem presentes neste grafo melhor será o resultado, porque o algoritmo usa somente as arestas que estão no complemento do grafo construído. Estas arestas representam o paralelismo existente na máquina para um dado programa. Em seguida integra-se o grafo de complemento com o grafo de interferência permitindo assim que o algoritmo de alocação leve em consideração o paralelismo existente. Com este novo grafo, efetua-se a alocação de registradores. O escalonador propriamente dito é ativado após a alocação.

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 algumas variáveis na memória. Tendo em vista estes fatos, o problema da alocação para arquiteturas superescalares é encontrar um mapeamento de registradores que faça uso de um número mínimo de registradores físicos, que o custo de derramamento seja mínimo e cujo grafo de escalonamento não possua dependências falsas. Para isto aplicam-se sobre o *grafo de interferência paralelizável* as mesmas heurísticas usadas tanto na alocação como no escalonamento. 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. Considerando a remoção de algumas arestas que evitam dependências falsas, está-se fazendo o trabalho do escalonador quando, devido a demanda por registradores, algumas opções de paralelismos 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 código[7].

Os algoritmos de Chaitin [Chaitin, 1982] e Pinter [Pinter, 1993] não discutem o problema de pares de registradores. Pares de registradores são muitas vezes necessários em processadores para representar duas metades de registradores usando precisão dupla para cargas e armazenamentos, para movimentação de registradores, etc. Em seus algoritmos, um pseudo-registrador v pode ser removido do grafo se for garantido existir um registrador físico para ele durante a fase de coloração. Isto significa que o seu grau, ou seja, o número de seus vizinhos, deve ser inferior ao número de registradores disponíveis. A necessidade por mais de um registrador para um dado vértice v muda a definição de seu grau. Um vértice v é considerado sem restrições quando a soma das necessidades de registradores físicos de seus vizinhos mais o número de registradores físicos necessários a ele for menor que o número de registradores alocáveis. Com esta modificação introduzida é possível obter uma coloração do *grafo de interferência paralelizável* mesmo quando a demanda dos vizinhos de um vértice v seja maior que o número de registradores físicos alocaveis. A reutilização de cores em vizinhos cujas arestas não se interferem neste grafo pode gerar uma coloração do grafo sem alterar o objetivo do algoritmo proposto por Pinter, que é obter uma ótima alocação de registradores sem introduzir dependências falsas no grafo de escalonamento.

4 A LDA

A linguagem de descrição de arquitetura, LDA, usada em GGCO é composta de três seções: seção de declarações, seção com as características da máquina alvo e seção de instruções. Na seção de declarações são especificados os registradores, os recursos da máquina, as constantes, o tamanho da memória disponível entre outras informações. Na seção referente a

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

máquina alvo são declarados os conjuntos de registradores de propósito geral, os argumentos, apontadores de pilha, área global, etc. A seção de instruções introduz a descrição de instruções da máquina, instruções especiais e transformações necessárias para casar padrões da linguagem intermediária com padrões da linguagem da máquina alvo (detalhes sobre a LDA são encontrados em [Bigonha and Rangel, 1994]).

A LDA ainda não cobre toda a classe das máquinas superescalares, contudo, ela é capaz de modelar a maior parte de suas características, abrangendo também as `RISCs`. Para ter uma idéia de sua aplicabilidade bem como do sistema GGCO como um todo destacamos os aspectos mais importantes destas arquiteturas ainda não cobertos por outros sistemas:
(1) A janela de registradores: a LDA oferece ao projetista do compilador facilidades para especificar separadamente os argumentos e parâmetros, o que modela a renomeação de registradores, e, também facilidades para modelar o salvamento dos registradores especificados pelo procedimento chamador e chamado.
(2) A descrição de máquina em LDA relaciona as necessidades de recursos para cada instrução da máquina alvo. A partir desta informação, GGCO constroi um vetor de recursos para cada instrução. Cada elemento deste vetor contém todos os recursos necessários à instrução em um determinado ciclo de máquina. Para evitar a ocorrência de uma dependência de recurso, o escalonador compara o vetor de recursos usados pela instrução candidata ao escalonamento com um vetor de recursos que é a composição de recursos necessários a todas as instruções correntemente em execução. Se a interseção for vazia, a instrução candidada é escalonada. Para solucionar dependências mais complicadas, como por exemplo, o esquema de prioridades por *hardware* para contenção de recursos, GGCO adota o seguinte esquema: (a) permite que uma série de prioridades seja associada com a "declaração de recursos" na descrição da máquina; (b) permite que um elemento do vetor de recursos seja associado com uma "instrução" para indicar sua prioridade; (c) examina prioridades durante a verificação por dependências de recursos e permite que escalonamentos contenham estas dependências, se eles forem causados por recursos prioritizados. Exemplo de um processador que possui este esquema é o Motorola MC88000. Nele, para controlar o uso de registrador no barramento destino, as instruções de inteiros têm uma prioridade mais alta, seguidas de instruções de ponto flutuante e então de instruções *load*.
(3) Para evitar as dependências de controle, a LDA especifica o número de *delay slots* dentro da diretiva de instrução. Para evitar o preenchimento dos *slots* com *no-ops* como acontece com o sistema Marion [Bradlee et al., 1991b] propusemos a implementação da técnica de Hennessy e Gross [Hennessy and Gross, 1983] em GGCO, incluíndo-a como um passo separado, intraprocedimental, após o escalonamento. Nesta técnica tenta-se preencher os *slots* com instruções válidas que estão antes da instrução de desvio, com instruções que seguem o objeto do desvio e com instruções que seguem a própria instrução de desvio. Gross e Hennessy mostram que arquiteturas cujas instruções de desvios possuem um *delay slot* que sempre é executado, seu algoritmo preenche 90% dos *slots* com instruções válidas.
(4) O sistema GGCO foi especificado para dar suporte ao despacho múltiplo de instruções particionando o conjunto de instruções de tal forma que uma instrução de cada partição pode ser despachada a cada ciclo.

(5) A maior parte das arquiteturas superescalares possuem instruções que usam o registrador de código de condição. A solução adotada neste caso é declarar o registrador de código de condição como sendo um registrador temporal[8] na seção de declarações da LDA. Declarar na seção de instruções as instruções que o modificam e examinar este registrador durante o escalonamento, garantindo que o mesmo está ligado e corretamente referenciado.

4.1 Mapeamento com a Linguagem Intermediária

Raramente existe um mapeamento direto entre uma linguagem intermediária e um conjunto de instruções da máquina alvo. GGCO incorpora em LDA dois mecanismos herdados de Marion que auxiliam o usuário a completar este mapeamento: (1) A transformação "glue" é normalmente empregada quando um operador da linguagem intermediária pode ser mapeado em duas ou três instruções de máquina. O exemplo da Figura 2 mostra que para comparar e desviar em uma dada arquitetura é necessário a utilização da instrução "subcc", que coloca o código de condição em um registrador de propósito geral e a instrução de desvio "be", que verifica a condição neste registrador e desvia de acordo com seu valor. Para mapear o operador da linguagem intermediária "==" nesta seqüência de duas instruções, uma transformação "glue" expande a subárvore "($1 == $2)" na subárvore "(($1 :: $2) ==)". O novo operador "::", presente na linguagem intermediária de GGCO, casará a instrução "subcc" e "==" casará a instrução "be".

```
%instr  subcc  r, r, r[0]   (; clk_cc)        %instr  be  #rlab
{cc = ($1 :: $2); }                          {if  cc == 0
[IF; ID; IE; IW;]                                  goto $1; }
(1, 1, 0)                                         [IF; ID; IE; ]
                                                  (1, 1, 1)

 %glue  r, #UCONST13
    {($1 == $2) ==> (($1 :: $2) == 0;}
```

Figura 2: Instruções em LDA

(2) As funções de escape (" *func") resolve situações em que são necessários transformações que manipulam metades de registradores e, em geral, permite ao usuário efetuar mapeamentos complicados sem afetar a geração eficiente de código. Por exemplo, permite escrever funções em linguagem C para produzir uma seqüência de instruções individuais

[8] Registrador temporal é um registrador que contém resultados intermediários de sub-operações.

escalonáveis. Estas funções podem e devem incluir somente chamadas às primitivas exportadas por GGCO. Estas primitivas criam, manipulam operandos e geram instruções.

5 Filosofia da Definição Formal de LDA

A definição formal da LDA segue o método denotacional para especificação de semântica. As descrições de arquitetura, que compreendem o domínio[9] sintático desta descrição formal, são apresentadas na forma de sintaxe abstrata. A Figura 3 mostra o mapeamento da descrição de arquitetura de máquinas para o seu respectivo gerador de código. Este mapeamento é definido pela função gen-mdc que define a semântica de uma descrição em LDA. O resultado final deste mapeamento é um trecho de programa em linguagem C, ou seja, é o mapeamento de uma descrição de arquitetura para o domínio Mdc, que constitui a parte dependente de máquina do gerador de código para a arquitetura descrita.

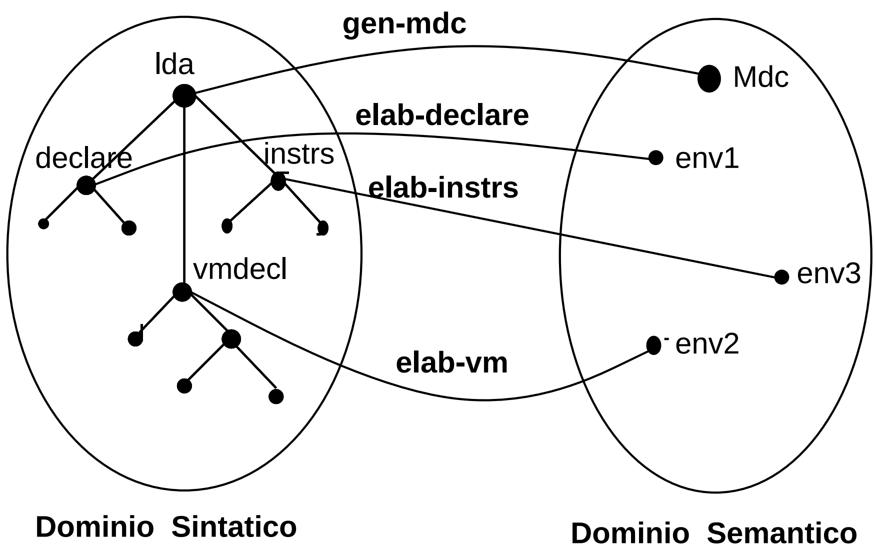

Figura 3: Mapeamento de lda em mdc.

A função gen-mdc recebe como entrada um arquivo com a especificação lda, na forma de uma árvore de sintaxe abstrata, ativa as funções `elab-declare`, `elab-vmdecl`, `elab-instr` e `elab-tables` para elaborar as várias partes da descrição de máquina. A função `elab-declare`

[9] Domínios podem ser vistos como sinônimos de tipo, podendo ser utilizados para modelar propriedades sintáticas ou semânticas em uma linguagem de programação. O domínio de um símbolo terminal é ele mesmo. O domínio de símbolos não-terminais podem ser inferidos, capitalizando sua primeira letra.

elabora as declarações de LDA; a função `elab-vmdecl` elabora as informações sobre a máquina alvo; a função `elab-instr` elabora as instruções de LDA, e a função `elab-tables` elabora as tabelas finais que constituem o código dependente de máquina do gerador de código. Na especificação deste mapeamento, destaca-se o domínio semântico "env" , que define o ambiente no qual as descrições formais são estabelecidas, ou seja, define uma tupla composta da identificação de todas as entidades auxiliares geradas para a elaboração das tabelas finais. Este domínio compreende um conjunto de tabelas e listas construídas a partir de cada especificação em LDA. Ele é parte da assinatura da maior parte das funções semânticas.

A presente definição formal está escrita em $\mathcal{SCRIPT}$ [Bigonha, 1994b]. $\mathcal{SCRIPT}$ é uma linguagem funcional que oferece uma notação simples para descrição da semântica denotacional de uma linguagem de programação de forma modular e legível. A sintaxe abstrata é definida em $\mathcal{SCRIPT}$ como uma lista de produções de uma gramática livre do contexto. Os não-terminais representam domínios sintáticos e os terminais são domínios de "quotations" , contendo apenas o elemento especificado. Os vários domínios semânticos da definição formal são agrupados em módulos, que controlam a visibilidade de suas denotações e provêem os serviços associados a cada domínio. Para cada um dos principais domínios semânticos usados na definição do mapeamento gen-mdc existe um módulo $\mathcal{SCRIPT}$, que encapsula as suas denotações e provêem os serviços para sua administração.

5.1 Principais Tabelas Geradas

A função de nível mais alto da definição formal de LDA, a função gen-mdc mapeia descrição lda em trechos de programas em C, os quais definem a porção dependente de arquitetura do gerador de código. Este trecho de programa, contido no arquivo MD.c, pressupõe as definições das estruturas de dados e tipos declarados no arquivo pré-definido MD.h (veja Seção 3).

As tabelas mais importantes são a tabela de recursos e a tabela de produções. A tabela de recursos descreve os recursos utilizados pelas instruções e constitui uma das informações mais importantes para o escalonamento. Seu conteúdo é usado essencialmente na rotina básica de escalonamento de instruções, para verificar os conflitos existentes e agrupar instruções.

A tabela de produções contém informações sobre as instruções. Cada linha desta tabela corresponde a uma diretiva de instrução fornecida na descrição da arquitetura em LDA e contém o seguinte: (1) uma árvore padrão e um símbolo de substituição derivado da expressão contida na diretiva; (2) uma matriz que indica para cada instrução, os tipos de seus operandos e suas localizações dentro do padrão e árvores objetos; (3) um índice para uma matriz de vetores de recursos e (4) dados referentes a latências, custo, *delay* *slot*, etc. Informações contidas na tabela de produções são utilizadas em várias funções do gerador de código. Por exemplo, nas rotinas de escalonamento, na construção do DAG de código, na construção do DAG de interferência de registradores, durante a coloração do grafo de interferência, no reconhecimento de padrões, etc. Outras estruturas de dados contêm declarações, informações da máquina virtual, classes, elementos, latências auxiliares, endereços das funções de escape e padrões de transformações "glue" .

As rotinas dependentes da arquitetura mais importantes são aquelas que retornam o conjunto de registradores de propósito geral para um dado tipo e aquela que contém informações relativas a sobreposição de registradores. Estas informações são utilizadas durante a criação dos *pseudo-registradores* e a alocação de registradores físicos.

6 Modelos de Pilha de GGCO

Uma informação importante para o projetista do compilador é o modelo de pilha utilizado na implementação de uma dada arquitetura. Quatro modelos de pilha são usados para descrever o modelo de execução da máquina alvo em GGCO. Estes modelos são definidos de acordo com a direção do par `(stack, frame)` apresentados abaixo. O par `(stack, frame)` é obtido diretamente das informações especificadas pelas diretivas que manipulam a pilha e o registro de ativação definidos na LDA [Bigonha and Rangel, 1994].

(down,down): Neste modelo a pilha cresce para baixo, o apontador do registro de ativação é usado para argumentos e variáveis locais; e a área derramada está entre as variáveis locais e o apontador da pilha.

(down,up): Neste modelo a pilha também cresce para baixo mas o apontador do registro de ativação é usado somente para argumentos e a área derramada está entre as variáveis locais e os argumentos.

(up,down): Neste modelo a pilha cresce para cima, o apontador do registro de ativação é usado somente para os argumentos e a área derramada está entre as variáveis locais e os argumentos.

(up,up): Neste modelo a pilha também cresce para cima mas o apontador para o registro de ativação é usado para variáveis locais e argumentos; a área derramada está entre as variáveis locais e o apontador da pilha.

As seguintes convenções são válidas para os quatro modelos apresentados: (1) argumentos de chamada utilizam o apontador da pilha; (2) argumentos de entrada são deslocamentos positivos a partir do apontador do registro de ativação; (3) registradores locais são deslocamentos negativos a partir do registro de ativação; (4) registradores globais são deslocamentos positivos a partir do apontador da área global.

A arquitetura SPARC não possui instruções explícitas ou registradores sobre pilhas, portanto o gerenciamento da pilha do usuário é resultado de algumas convenções de software. Por exemplo, (1) na criação de um processo, o sistema operacional reserva uma área para a pilha. Um registrador específico representado por "%sp" aponta para seu topo (posição mais baixa) e um outro registrador específico representado por "%fp" aponta para o registro de ativação corrente; (2) a pilha cresce a partir do endereço de mais alta ordem para o endereço de mais baixa ordem; (3) alguns registradores são preservados entre troca de janelas; (4) a área reservada para tratar estouros deve iniciar na posição indicada por "%sp" mesmo quando a altura da pilha variar devido a alocação dinâmica de memória; (5) extensão da pilha é de 64 *bytes*. Os tratadores de estouro pressupõem um alinhamento em múltiplos de oito *bytes*. Decorre desta convenção que os parâmetros de uma função são encontrados a partir do endereço "fp + 64" , as variáveis locais e automáticas presentes na pilha são endereçadas em relação ao "%fp" . Temporários e parâmetros de saída são endereçados especificando um deslocamento positivo a partir do "%sp" corrente ("%sp + 64") [Souza, 1992].

Janela de registradores simula a estrutura de pilha da SPARC com duas diferenças: (1) a estrutura é implementada nos registradores do "chip" ; (2) o tamanho das áreas para parâmetros de entrada, variáveis locais e parâmetros de saída são fixos. Muito embora as vantagens desta abordagem em relação a abordagem de GGCO sejam significativas: (1) não é necessário referência à memória para se ter acesso aos dados da pilha; (2) dois operandos podem ser lidos e um terceiro escrito a cada ciclo de máquina porque o dado é residente no arquivo de registradores. A utilização de janelas de registradores em sistemas multiprocessados deve ser analisada tanto do ponto de vista de um processo quanto do sistema. Em tais sistemas, como é o caso de sistema operacional "UNIX" , existem vários processos executando simultaneamente (pseudo-paralelismo). Para que cada processo tenha acesso aos recursos físicos do ambiente, por exemplo, "CPU" , o sistema operacional realiza a troca do contexto de um processo para o de um outro. Caso o processo que estava executando anteriormente, tenha utilizado todas as janelas de registradores, é necessário salvar e invalidar todas janelas, inclusive aquelas que tiveram que ser alocadas na memória para serem utilizadas por outros processos. Desta forma pode ocorrer um aumento no *overhead* de troca de contexto, o que caracteriza uma grande desvantagem do uso de janelas de registradores. Uma opção de utilização das janelas de registradores em tais sistemas seria na base de uma por processo. Neste caso, em um sistema de multiprocesso poderia ter, por exemplo, até oito processos sendo executados, simultaneamente, cada um deles com sua janela de registradores. Se o processador parasse de executar um processo e passasse a executar outro, não seria necessário recarregar todo o seu contexto, o sistema operacional simplesmente trocaria a janela de registradores para a daquele processo e continuaria executando. De certa forma o fato de conceder todo recurso para cada processo ou compartilhá-lo com outros processos é uma escolha do projetista do compilador. Achamos que janela de registradores auxilia do ponto de vista de chamada de função mas implica em uma transferência muito grande de trabalho para este projetista. O ganho é mais real em um sistema monoprocessado, contudo, ele é questionável em sistemas de multiprocessos.

A pilha do sistema GGCO é uma pilha clássica, ela guarda parâmetros passados, variáveis automáticas e endereço de retorno. Informação suficiente para cobrir grande parte das arquiteturas superescalares. Basicamente, pelo fato das janelas de registradores simularem a estrutura de pilha da SPARC, a pilha do usuário do sistema GGCO deveria fazer uma série de mudanças para tentar acomodá-la. Por exemplo, entre outras informações, ela deveria possuir uma área para salvar os registradores de entrada e locais no caso de estourar a pilha de janelas. Outra informação relevante é o tipo do parâmetro passado, existem algumas regras que têm que ser observadas quanto ao local onde os mesmos são colocados. Se o valor de retorno é um inteiro, este deve ser colocado em um dos seis registradores para inteiro; se o valor não for um inteiro ou real, retorna-se o endereço, o que acarreta por sua vez em uma série de providências; no caso do real, o valor retornado é colocado nos registradores de ponto flutuante. Mas a diferença principal não está aqui, e sim nos algoritmos que manipulam a pilha. No projeto de GGCO pensou-se inicialmente em utilizar a pilha da SPARC mas, chegou-se a conclusão de que se o fizesse, estaria projetanto um sistema totalmente preso a sua arquitetura. Para executar uma especificação, por exemplo, da arquitetura MC88000 que não possui a facilidade de janela, os algoritmos projetados para manipulá-la não serveriam para nada. Portanto, um sistema com este modelo de pilha não é genérico, pelo contrário, é totalmente ligado a uma máquina específica, inclusive se isto traz um ganho, traz um ganho somente naquele processador porque em máquinas que não possuem esta característica, o ganho seria nulo.

Para implementar um sistema que mantenha estas duas estruturas paralelas, uma forma seria manter os quatros modelos descritos e projetar a pilha da SPARC de tal forma que quando fosse compilar para a SPARC, poderia ser utilizado recurso de pré-processamento para ter acesso ao arquivo onde estaria definido sua pilha. Contudo, esta abordagem resultaria em um sistema ineficiente. Optamos por projetar um novo sistema que seja um superconjunto deste. A parte do sistema GGCO implementado ainda não modela a pilha que engloba janela de registradores, contudo LDA possui as diretivas para tal. Para incorporar janela de registradores, talvez uma opção seja implementar o algoritmo de renomeação, porque na verdade, a troca de janelas salva e renomeia os registradores na ativação de um procedimento: os registradores declarados como saída passam a ser entradas do novo procedimento.

7 Conclusões

Neste trabalho apresentamos uma ferramenta de auxílio à construção de geradores de código otimizados cujas principais contribuições englobam o projeto de um sistema baseado na descrição de arquiteturas que incorpora, de forma que consideramos satisfatória, uma linguagem para descrever as arquiteturas que contém restrições sobre o escalonamento. Esta linguagem está formalmente especificada. Sua definição formal pode ser vista como um sistema gerador de gerador de código a partir da descrição da arquitetura e, cujo resultado final é um trecho de programa em linguagem C, que constitui a parte dependente de máquina do gerador de código para o processador descrito. Identificamos, também, o nível de interdependência satisfatório entre o escalonamento de instruções e a alocação de registradores.

O sistema GGCO descrito neste trabalho ainda não está totalmente implementado, contudo a especificação da parte dependente de arquitetura encontra-se definida em [Bigonha, 1994a]. Fazem parte da continuação deste trabalho um estudo e implementação de um esquema para modelar múltiplas unidades funcionais idênticas e a implementação e avaliação do algoritmo de alocação de registradores apresentado informalmente na Seção 3.2.3;

Referências

[Benitez and Davidson, 1988] Benitez, M. E. and Davidson, J. W. (1988). A portable global optimizer and linker. *ACM Sigplan Notices*, 23(7).

[Bigonha, 1994a] Bigonha, M. A. S. (1994a). *Otimização de Código em Máquinas Superescalares*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática-PUC/RJ.

[Bigonha and Rangel, 1994] Bigonha, M. A. S. and Rangel, J. L. M. (1994). Linguagens para descrição de arquiteturas de computadores. In *(a ser publicado)*.

[Bigonha, 1994b] Bigonha, R. S. (1994b). $\mathcal{SCRIPT}$ an object oriented language for denotational semantics (user's manual and reference). Relatório Técnico 03/94, Departamento de Ciência da Computação, UFMG.

[Bradlee et al., 1991a] Bradlee, D. G., Eggers, S. J., and Henry, R. R. (1991a). 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.

[Bradlee et al., 1991b] Bradlee, D. G., Eggers, S. J., and Henry, R. R. (1991b). The marion system for retargetable instruction scheduling. In *ACM Sigplan Conference on Programming Language Design and Implementation*. ACM Sigplan Notices 26(7).

[Chaitin, 1982] Chaitin, G. J. (1982). Register allocation and spilling via graph coloring. *ACM Sigplan Notices*, 17(6). ACM Sigplan Symposium on Compiler Construction, 1982.

[Costa, 1990] Costa, P. S. S. (1990). Um gerador automático de geradores de código. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro.

[Davidson, 1981] Davidson, J. W. (1981). *Simplifying Code Generation Through Peephole Optimization*. PhD thesis, Department of Computer Science - The University of Arizona, Tucson Arizona.

[Fisher et al., 1984] Fisher, J. A., Ellis, J. R., Ruttenberg, J. C., and Nicolau, A. (1984). Parallel processing: A smart compiler and a dump machine. *ACM Sigplan Notices*, 19(6). Proceedings of the ACM Sigplan, Symposium on Compiler Construction, 1984.

[Fraser and Hanson, 1992] Fraser, C. W. and Hanson, D. R. (1992). A code generation interface for ansi c. Research Report CS-TR-270-90, Department of Computer Science, Princeton University. Last Revised September 1992.

[Gibbons et al., 1986] Gibbons, P. B., , and Muchnick, S. S. (1986). Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction*. ACM Sigplan Notices 21(7).

[Goodman and Wei-Chung-Hsu, 1988] Goodman, J. R. and Wei-Chung-Hsu (1988). Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing - Conference ACM-PRESS Proceedings*, St. Malo France.

[Hennessy and Gross, 1983] Hennessy, J. and Gross, T. (1983). Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3).

[Henry, 1987] Henry, R. R. (1987). Code generation by table lookup. Technical Report # 87-07-07, Computer Science Department, University of Washington, FR-35 Seattle, WA 89195 USA.

[Intel, 1989] Intel, I., editor (1989). *i860 64-bit Microprocessor Programmer's Reference Manual*. Intel, Inc., Santa Clara, California.

[Kerns, 1993] Kerns, D. R. (1993). Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation*, Albuquerque NM. ACM Sigplan Notices - Vol. 28 number 6 june-1993.

[Motorola, 1984] Motorola, I., editor (1984). *MC68020 32-Bit Microprocessor User's Manual*. Prentice-Hall, Englewood Cliffs, New Jersey.

[Motorola, 1990] Motorola, I., editor (1990). *MC88100 RISC Microprocessor User's Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition.

[Pinter, 1993] Pinter, S. S. (1993). Register allocation with instruction scheduling: a New Approach. In *Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation*, Albuquerque NM. ACM Sigplan Notices - Vol. 28 number 6 june-1993.

[Souza, 1992] Souza, G. B. (1992). Técnicas de otimização de código para arquiteturas RISC. Master's thesis, Departamento de Ciência da Computação, UNICAMP Campinas, SP.

[Stallman, 1989] Stallman, R. M. (1989). Using and porting GNU C. Free Software Foundation Incorporation. Cambridge Massachusetts.

[Tiemann, 1989] Tiemann, M. D. (1989). The gnu instruction scheduler. Class Report CS 343, Stanford University.

[Warren Jr, 1990] Warren Jr, H. S. (1990). Instruction scheduling for the ibm risc system/6000 processor. *IBM Journal of Research and Development*, 34(1).