Linguagens para Descrição de Arquiteturas de Computadores e Geradores de Código

LLP05/99

Mariza A. S. Bigonha

13 de maio de 1999

Abstract

This paper presents a comparative study of different description languages for real computer architecture found in the literature. It begins with the presentation of some code generation and optimization systems that make use of machine architecture description. It also presents a brief description of a notation for resgister transfer used in the instructions description in one of the approaches presented here, and shows how this notation can be used to describe machine architecture. In addition, it is presented and evaluated some other notations used in the description of machines. The conclusion drawn from this work is that even though it is difficult to achieve consensus about which language is the most adequate, some characteristics of the language help in the selection process.

Keywords: Superscalar architecture, architeture description language, code generation.

Resumo

Este relatório apresenta um estudo comparativo das diversas linguagens existentes para descrever arquiteturas de máquinas reais. Ele inicia com a apresentação de alguns sistemas de geração de otimização de código que se utilizam de descrições de arquiteturas de máquinas alvo para atingirem seus objetivos. Introduz uma notação para transferência de registradores utilizada para descrever as instruções em uma das abordagens apresentadas neste texto e mostra como arquiteturas de máquinas podem ser descritas nesta notação. Também são apresentadas, por meio de exemplos, algumas notações utilizadas nas especificações de máquinas, seguida de uma análise das mesmas. Desta análise, conclui-se que, muito embora seja difícil chegar-se a um consendo sobre qual linguagem é mais adequada, certas características presentes ou ausentes em algumas linguagens auxiliam no processo de seleção.

Palavras-chave: Arquiteturas CISC e RISC, linguagens descrição arquitetura, geração de código.

Sumário

1 Introdução

Qualquer discussão sobre geração de código deve considerar a arquitetura alvo e, isso introduz a necessidade de um mecanismo adequado para especificá-la. As linguagens de descrição de arquiteturas (LDA) desempenham este papel. Muito embora sejam encontrados na literatura sistemas geradores de geradores de código que as utilizam, as linguagens existentes não descrevem os principais processadores e, quando o fazem, não são capazes de especificar suas características mais complexas. Uma das qualidades mais almejadas em uma linguagem para descrição de arquitetura para um sistema redirecionável é a completeza. Completeza significa que a linguagem é capaz de modelar todas as máquinas existentes dentro de uma classe particular.

Nas arquiteturas tradicionais, o conjunto de instruções é caracterizado por sua complexidade, valhendo-lhes o nome CISC (*Complex Instruction Set Computers*). Nas arquiteturas mais recentes o conjunto de instruções é caracterizado pela simplicidade, cabendo-lhes o nome de RISC (*Reduced Instruction Set Computers*). Muito embora exista um grande volume de publicações relacionadas com as arquiteturas CISC e RISC, ainda não foi desenvolvida uma teoria bem fundamentada, como a que existe para *front-ends* de compiladores, sobre a qual possam se basear os projetistas de *back-ends*, especialmente aqueles voltados para arquiteturas superescalares. As arquiteturas superescalares são uma evolução dos processadores RISC. Estas arquiteturas 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. Percebe-se, inclusive, que há discordância entre pesquisadores da área em relação a determinados enfoques. Por exemplo, do ponto de vista das linguagens de descrição de arquiteturas, existem duas correntes. Uma delas defende o princípio de que devem ser incorporadas às linguagens primitivas para auxiliar o escalonamento de instruções[1] [Wall and Powell, 1987, Bradlee et al., 1991]. A outra corrente defende o princípio de que estas primitivas não precisam ser incorporadas explicitamente nas linguagens de descrição de arquiteturas para que o escalonamento de instruções seja feito de forma eficaz [Benitez and Davidson, 1988, Stallman, 1989, Benitez and Davidson, 1991].

Este relatório apresenta um estudo comparativo das diversas linguagens existentes para descrever arquiteturas de máquinas CISCs e RISCs reais. A Seção 2 apresenta uma revisão da literatura dos sistemas de geração de código redirecionáveis para arquiteturas CISC que se utilizam de descrições de arquiteturas de máquinas alvo para atingirem seus objetivos. A Seção 2.6 introduz uma linguagem para descrever as instruções em uma das abordagens apresentadas neste texto. Também são apresentadas na Seção 2.7, por meio de exemplos,

[1] Escalonamento de instruções é uma técnica de *software* que rearranja seqüências de código durante a compilação com o objetivo de reduzir possíveis atrasos de execução.

algumas notações utilizadas nas especificações das máquinas, seguidas de uma análise das mesmas. Desta análise, conclui-se que, muito embora seja difícil chegar a um consenso sobre qual linguagem é a mais adequada, certas características presentes ou ausentes em algumas linguagens auxiliam no processo de seleção. A Seção 3 apresenta as abordagens existentes para as arquiteturas RISCs. O relatório conclui apresentando as propriedades desejáveis em uma LDA para arquiteturas CISC e que podem ser uteis para descrição de arquiteturas RISC. Em particular, mostra brevemente sistemas que tentam modelar a técnica de escalonamento de instruções, cobrindo assim o que existe para arquiteturas superescalares.

2 Geradores de Geradores de Código para Arquiteturas CISC

Iniciamos apresentando os principais trabalhos dentro de quatro famílias de geradores de código (veja Figura 1), encontrados atualmente na literatura, que se utilizam de linguagens de descrição de arquiteturas para obter o resultado almejado.

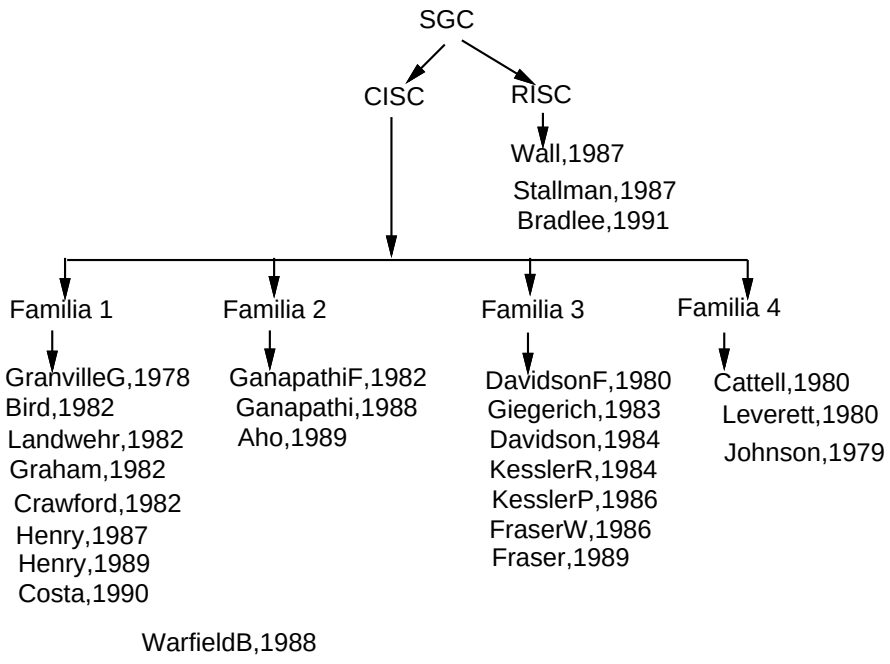


Figura 1: Uso de LDAs em Sistemas de Geração de Código.

2.1 Primeira Família de Geradores de Código

Na primeira família de geradores de código têm-se entre outros o sistema de Granville e Graham [Graham and Glanville, 1978] [Graham et al., 1982], o CODEGEN de Henry [Aigrain et al., 1984, Henry and Damron, 1989a, Henry and Damron, 1989b] e o sistema AutoCode de Costa [Costa, 1990].

2.1.1 Abordagem de Glanville

Glanville e Graham [Graham and Glanville, 1978, Graham et al., 1982] construíram o primeiro sistema redirecionável de gerador de código. Seu sistema tem como entrada a descrição da máquina em forma de gramática e produz como resultado um gerador de código que utiliza técnicas de análise sintática LR para efetuar o casamento de padrão com a linguagem intermediária. As produções da gramática são compostas de padrões, que englobam os símbolos terminais e não-terminais, e de um lado esquerdo que é formado por um símbolo não-terminal. Ações associadas às produções dirigem a geração do código de máquina.

Construir tabelas para geração de código é similar à construção de tabelas para a análise sintática a partir de uma gramática. De fato, a teoria utilizada na análise sintática baseada em gramáticas livres do contexte foram utilizadas no desenvolvimento do algoritmo. como resultado, o algoritmo é rápido e de fácil entendimento.

O gerador de cópdigo produz um bom código local. Em sua dissertação de doutorado, *A Machine Independent Algorithm for Codoe Generation and its use in Retargetable Compilers*, Glanville discute a implementação de geradores de código para duas mquinas, o PDP-11 e o IBM 370. Muito embora estas máquinas possuam arquiteturas distintas, é difícil avaliar a habilidade do algoritmo de se adaptar a um universo maior de arquiteturas. Glanville supõe ter dificuldades com a adaptação de seu método com a família CDC Cyber devido a sua arquitetura "desajeitada".

Um problema mais sério aparece na descrição de máquina. Devido ao fato das tabelas serem construídas usando a técnica de análise sintática baseada em gramáticas livres do contexto, é necessário descrever cada instrução com todos os modos de endereçamento possível. Por exemplo, o `PDP11` possui oito modos de endereçamento para uma palavra. Em algumas instruções de operando duplo, todo modo de endereçamento pode ser utilizado no fonte e no destino. Para uma instrução *mov* por exemplo, existem sessenta e quatro combinações de modos de endereçamento possíveis. Isto significa que uma descrição completa do `PDP11` deve conter sessenta e quatro padrões somente para descrever a instrução *mov*.

2.1.2 Abordagem de Henry

Robert Henry [Aigrain et al., 1984, Henry and Damron, 1989a, Henry and Damron, 1989b] descreve um gerador de código redirecionável, denominado *CODEGEN*, projetado para substituir *pcc1*, o gerador de código da primeira versão do Compilador C. *CODEGEN* é um sucessor do sistema de Graham-Glanville. Desde de 1980, tem sido utilizado para estudar a organização de geradores de código, as linguagens de especificações de geradores de código, os esquemas de transformações de árvores, os geradores de reconhecedores de padrões em árvores, a administração de registradores, os otimizadores locais e as técnicas utilizadas para descrever arquiteturas de máquinas.

CODEGEN é somente uma parte de um compilador completo, contendo um *front-end*, o próprio *CODEGEN* um montador e um editor de linquedição.

CODEGEN consiste de quatro fases lógicas: o transformador de árvores, o seletor de instruções, o administrador de temporários e o formatador de instruções. A estrutura do gerador de código é análoga ao modelo utilizado em um compilador de um único passo dirigido por sintaxe. O transformador de árvores desenvolve o papel de um analisador léxico, o seletor de instruções desempenha o papel de um analisador sintático; e o administrador de temporários e o formatador de instruções (EMIT) desempenham o papel do atribuidor. O transformador de árvores lê árvores na forma intermediária e as converte em uma floresta de pequenas árvores expressas em uma notação um pouco diferente da representação intermediária. O seletor de instruções é responsável pela determinação de seqüências de instruções de máquina semânticamente equivalentes a uma árvore na forma intermediária. O administrador de temporários é responsável pela alocação e liberação de temporários, tanto em registradores como na memória principal. O formatador de instruções é responsável pelo agrupamento de fragmentos de instruções virtuais, convertendo-as em instruções de máquina alvo. O adminstrador de temporários e o formatador de instruções manipulam atributos descrevendo partes das instruções.

O seletor de instruções é redirecionado para outra máquina, substituindo-se a especificação das instruções da máquina alvo. A descrição das instruções é definida como uma enumeração de cinco-tuplas denominada *PPRACs* ou *regras*. Um conjunto de *PPRACs* em uma descrição de máquina é denominada por *P*. Cada *PPRAC* modela a semântica de uma parte ou de toda instrução da máquina alvo. Uma *PPRAC* é constituida de um padrão, predicado, substituição (*replacement*), ação e um custo.

Um *padrão* deve ser uma árvore e pode conter símbolos de substituição somente nas folhas. O *predicado* deve ser uma função booleana que descreve quais condições semânticas devem ser válidas antes que uma *PPRAC* possa casar. Se o *predicado* for omitido, assume-se verdadeiro. A *substituição* deve ser um símbolo não-terminal. A *ação* é um trecho de código na linguagem de implementação. Cada um dos C_i deve ser uma função retornando

um número não negativo. O C_i modela o custo do i-ésimo recurso.

Uma *PPRAC* é definida da seguinte forma:

replacement $\rightarrow$ *pattern* `when` *predicate* = *action* `cost(`$C_1 \cdots C_{ncost}$`)`

analisada como:

(*replacement* $\rightarrow$ *pattern*) (`when` *predicate)* = *(action*) (`cost(`$C_1 \cdots C_{ncost}$`)`) Ξ

e lida da seguinte forma: se um padrão casa, e se o predicado for verdadeiro, e se a *PPRAC* for selecionada, então a ação semântica é avaliada por seu efeito colateral, a porção da subárvore casada é reescrita com a substituição (*replacement*), e o custo incurrido é descrito pelo componente de custo.

Resumindo, PPRACs se assemelham às produções do sistema de Graham-Glanville, mas são acrescidas de predicados e custos para dirigir o casamento (*matching*). O casador (*matcher*) pode utilizar diferentes técnicas, incluindo analisadores LR, analisadores sintáticos descendentes recursivos, casamento de descendentes (*top-down matching*) e casamento de ascendentes (*bottom-up matching*). A escolha de uma destas técnicas afeta sensivelmente a eficiência do código gerado, o tempo de construção do compilador, a velocidade do compilador e o tamanho das tabelas ou códigos. O projetista do compilador deve selecionar a técnica que melhor se adapte ao seu sistema. CODEGEN também inclui o *compiler writer directed tree transformer* que auxilia no mapeamento da linguagem intermediária para o código da máquina alvo. A maior vantagem do sistema CODEGEN está no fato de que virtualmente todos os aspectos da arquitetura da máquina podem ser capturados em uma única abordagem, na gramática. A desvantagem é que é difícil capturar alguns efeitos colaterais, como por exemplo, endereçamento auto-incremento. Além desta desvantagem, as tabelas que dirigem o casamento e reduções podem ser muito grandes. Contudo, Fraser e Henry, em seu trabalho [Fraser and Henry, 1991], conseguiram minimizar este problema com sucesso.

2.1.3 Abordagem de Costa

O trabalho de Costa [Costa, 1990], intitulado *AutoCode*, é um sistema de produção de geradores de código baseado em reconhecimento de padrões e dirigido por tabelas geradas automaticamente, a partir de uma descrição da arquitetura da máquina alvo. Seu sistema se assemelha à abordagem de Glanville e Graham.

Costa introduz três tipos de extensões à gramática de descrição da máquina alvo de Glanville. A primeira diz respeito à correspondência entre instruções e produções. Na aborda-

gem de Graham-Glanville, cada produção é associada a uma única instrução. Em *AutoCode*, cada produção em uma descrição de máquina pode ser associada a várias instruções, limitado apenas pela disponibilidade de memória da máquina onde o sistema esteja sendo executado. Esta extensão permite que instruções especiais sejam descritas com maior precisão. A segunda extensão diz respeito ao uso de operadores semânticos, que foram introduzidos com o objetivo de efetuar intervenções semânticas em tempo de geração de código [2]. A terceira e última extensão diz respeito às técnicas de fatoração gramatical. Dois tipos de fatoração foram incluídos: fatoração de modos de endereçamento e fatoração de operadores. A fatoração gramatical é uma forma de permitir que partes comuns a vários padrões sejam fatorados ao invés de duplicados nas produções. Por exemplo, para a fatoração de modos de endereçamento, a linguagem de AutoCode possui uma seção especial destinada à declaração de não-terminais, que condensa os diversos modos de endereçamento presentes na máquina alvo. Cada não-terminal declarado é associado, através de uma ou mais produções, a determinado padrão descrevendo um (trecho de) modo de endereçamento. Fatoração de operadores é a substituição de vários operadores por um único não-terminal de operação. Por exemplo, operadores como: *add, mul, and or* e *xor* podem ser agrupados em *binOp*. Isto é possível porque todos estes operadores possuem as mesmas características. Eles são binários, comutativos, possuem a mesma sintaxe e a semântica de seus operandos é idêntica. Assim, *binOp* pode substituir qualquer um dos operadores acima nas regras em que os mesmos executam a operação primária da instrução a ser emitida.

AutoCode é modularmente dividido em duas partes: a primeira parte refere-se à construção automática das tabelas que guiarão o algoritmo do gerador de código, e a segunda parte refere-se à geração de código propriamente dita.

A abordagem de Costa possui, entretanto, algumas características indesejáveis [Costa, 1990]. A primeira delas é a dependência de máquina da representação intermediária emitida pelo analisador sintático. Isto prejudica a portabilidade do compilador. A segunda diz respeito à inclusão de operadores semânticos na linguagem de descrição de arquitetura. Muito embora esta inclusão aumente o poder de expressão de uma descrição de máquina, por outro lado, aumenta também o tempo gasto para a geração de código, porque novas reduções devem ser efetuadas.

O esquema proposto por Costa pode mostrar-se ainda ineficiente, se aplicado a arquiteturas pouco simétricas. Modos de endereçamento e instruções complexas de máquinas especializadas criam, normalmente, um número indesejado de casos especiais a serem explorados pela gramática descritiva. Ademais, muitas arquiteturas possuem registradores cujo uso é limitado a operações exclusivas, e diversas instruções possuem efeitos colaterais difíceis de

[2] Intervenção semântica é uma técnica utilizada para identificar símbolos da gramática descritiva sempre que houver alguma equivalência entre dois símbolos na mesma instrução. Normalmente uma qualificação semântica é especificada por um "." seguida de um número inteiro.

serem formalizados. Estes fatores podem comprometer a qualidade do código gerado.

2.2 Segunda Família de Geradores de Código

Na segunda família de geradores de código têm-se o sistema de Ganapathi e Fisher [Ganapathi and Fischer, 1992, Ganapathi and Fischer, 1985] e o sistema de Aho, Ganapathi e Tjiang [Aho et al., 1989].

2.2.1 Abordagem de Ganapathi e Fisher

O trabalho de Ganapathi [Ganapathi and Fischer, 1992] é similar a abordagem de Glanville. Enquanto Glanville se utiliza de gramáticas Lr, Ganapath usa gramática de atributos. Na abordagem de Ganapathi, o algoritmo básico de Glanville é modificado para prover um processamento de atributos formalizado e um guia semântico automático. As principais extensões introduzidas estão relacionadas com a estrutura do gerador de código, atribuição de endereços, otimizações dependentes de máquina e redirecionamento. Atributos semânticos e atributos predicados são utilizados para atingir este objetivo. A máquina alvo é descrita usando gramática de atributos ao invés de gramática livre do contexto e a geração de código é efetuada pelo analisador sintático com a avaliação dos atributos.

Ganapathi produziu geradores de código para as arquiteturas `PDP-11, VAX-11/780`. Estas máquinas são tão similares queé impossível avaliar a habilidade de sua técnica para acomodar outras arquiteturas. A descrição da máquina é muito longa e difícil de ser entendida. A descrição de Ganapathi para o `PDP-11` possui sete páginas, enquanto que a descrição de Glanville possui três páginas. Da mesma forma, a descrição de Ganapathi para o `VAX-11/780` é composta de onze páginas.

Ganapathi se utiliza dos atributos para efetuar várias otimizações dependentes de máquina. Estas otimizações são manualmente embutidas nas ações semânticas da descrição da máquina. Conseqüentemente, a qualidade do código gerado é bem melhor do que os métodos existentes até então.

As vantagens atribuídas ao sistema de Ganapathi e Fischer são o tamanho reduzido das tabelas do analisador e a facilidade de capturar os efeitos colaterais. A desvantagem é que o projetista do compilador deve utilizar duas técnicas: gramática e regras de atributos. Um descendente deste sistema [Aho et al., 1989] utiliza programação dinâmica[3] com o

[3]O princípio do algoritmo de programação dinâmica é particionar o problema da geração de código "ótimo" para uma expressão em sub-problemas para a geração de código "ótimo" para as sub-expressões da expressão dada. Exemplo, dada a expressão E da forma $E_1 + E_2$, um "ótimo" programa para E é

algoritmo de reescrita de árvore para gerar código [Aho et al., 1989].

2.3 Terceira Família de Geradores de Código

Na terceira família incluem-se: (1) o sistema PO desenvolvido por Davidson e Fraser [Davidson and Fraser, 1980], também descrito em [Davidson W. and Fraser W., 1984], [Fraser W. and Wendt, 1986] e [Davidson and Fraser, 1984]; (2) o sistema PEEP desenvolvido por Robert Kessler [Kessler, 1984]; (3) o sistema desenvolvido por Peter Kessler [Kessler, 1986]; e (4) o sistema de Fraser [Fraser, 1989].

2.3.1 Abordagem de Davidson

Davidson e Fraser apresentam uma abordagem diferente das outras existentes até então para geração de código em arquiteturas CISCs. Códigos de máquinas simples, porém corretos, são gerados e então otimizados utilizando um otimizador local, PO, produzido a partir da descrição da arquitetura da máquina. Dado um programa em linguagem de montagem e uma descrição simbólica da máquina, PO simula pares de instruções adjacentes, e, quando possível, troca-as por uma única instrução equivalente. PO é composto de três fases distintas: *Cacher, Combiner* e *Assigner*. Cada uma destas fases opera com listas de transferências de registradores ("RTLs") que descrevem o efeito das instruções. *Cacher* determina o efeito de cada instrução. *Combiner* junta pares com o mesmo efeito e seleciona a instrução mais barata. *Assigner* atribui registradores e traduz a RTL otimizada para a linguagem de montagem. PO se utiliza da descrição da máquina para verificar a validade do resultado obtido. Quando PO termina, nenhuma instrução, ou par de instruções adjacentes pode ser substituída por outra de menor custo. Como resultado desta organização, PO é independente de máquina e pode ser descrito formalmente e concisamente.

A arquitetura de máquina para Davidson é descrita por meio de uma gramática para tradução dirigida por sintaxe entre a linguagem de montagem da máquina alvo e a transferência de registradores. A partir da descrição da máquina, um reconhecedor e um tradutor são construídos automaticamente. PO utiliza-se do reconhecedor para verificar se as transferências de registradores representam instruções válidas na máquina alvo. PO usa o tradutor para verter a representação interna das instruções para a linguagem de montagem da máquina alvo. Detalhes sobre a descrição da máquina são apresentados na Seção 2.6.2.

A descrição de máquina para PO é mais simples que a descrição da família de Graham-Glanville, permitindo um fácil redirecionamento. A desvantagem atribuída a PO é a falta

formado pela combinação de "ótimos" programas para E_1 e E_2, em qualquer ordem, seguida de código para avaliar "+"

de flexibilidade na escolha entre a velocidade do compilador e o tamanho do código ou tabelas. O sistema PEEP de Robert R. Kessler [Kessler, 1984] e o sistema de Peter B. Kessler [Kessler, 1986] são similares ao sistema PO de Davidson e Fraser.

Davidson e Fraser estendem *PO* para gerar automaticamente padrões ou regras que descrevem as otimizações que devem ser efetuadas. Um conjunto fixo de regras é gerado em tempo de geração do compilador e carregado em um otimizador dirigido por regras, intitulado *HOP*, (veja Seção 2.3.5).

A vantagem desta abordagem é que as regras são derivadas automaticamente. A desvantagem é que essencialmente deve se construir dois compiladores, um que opera usando a descrição da máquina e um outro que usa as regras. O segundo, entretanto, é construído automanticamente, uma vez que o primeiro é completado. O resultado final é sem dúvida um otimizador local rápido.

Davidson estendeu também PO para efetuar reorganização de instruções, incluindo *targeting* e determinação da ordem de avaliação, para reduzir o uso de registradores [Davidson, 1986]. O sistema foi projetado, inicialmente, para arquiteturas CISCs, mas para a arquitetura RISC denominada PRIMUS90, a reorganização de instruções também considerou os atrasos na *pipeline*. Contudo, a função para verificar os atrasos é feita manualmente e nenhuma informação sobre escalonamento é derivada da especificação da máquina.

2.3.2 Abordagem de Robert Kessler

Robert R. Kessler [Kessler, 1984] descreve um sistema, denominado *PEEP*, que, ao invés de analisar as sequências de instruções que ocorrem durante a geração de código, analisa a descrição da máquina durante a construção do compilador. A técnica proposta por ele limita-se a descobrir instruções que sejam equivalentes a sequências de instruções de comprimento dois. Kessler utiliza-se da descrição da máquina para encontrar todas as otimizações possíveis. Os efeitos de um par de instrução são combinados, e a descrição de instruções é pesquisada para descobrir uma única, mais eficiente e que tenha o mesmo efeito que as duas instruções combinadas. *PEEP* se apresenta em duas partes, o gerador de tabelas *PEEP* propriamente dito, que efetua a análise da máquina alvo, e o otimizador *PEEP* que efetua as otimizações como especificadas pelas tabelas produzidas pelo gerador de tabelas.

Robert Kessler utiliza-se de uma linguagem baseada em *LISP* para descrever a arquitetura da máquina alvo. Esta proximidade de *LISP* facilita-lhe a expressão de construções e proporciona uma grande flexibilidade na escrita das definições, além de permitir ao usuário escrever macros *LISP*, quando necessário. O usuário pode definir constantes, registradores, modos de endereçamento e instruções. Na maioria das arquiteturas, cada instrução permite

vários modos de endereçamento para cada operando. A linguagem proposta por Kessler permite a definição de uma enumeração na especificação de cada um dos operandos, ou seja, os operandos das instruções são definidos como um conjunto de todos os modos de endereçamento possíveis. Esta enumeração é utilizada durante o casamento de padrões de instruções. Ao pesquisar por uma otimização, dois operandos podem ser interceptados para verificar a superposição de modos de endereçamentos. Se a interseção é vazia, a otimização é ignorada. As instruções são descritas provendo o seu formato de entrada, as equações semânticas definindo suas funções e o seu custo englobando tempo e espaço.

2.3.3 Abordagem de Peter Kessler

Para aliviar alguns dos problemas inerentes a um modelo dirigido puramente pela sintaxe, Peter B. Kessler [Kessler, 1986] sugere a inclusão de uma fase separada de transformação de código. Assim, construções de propósito especial são completamente removidas da descrição da máquina. E, ao invés de usar a composição "força bruta" para formar sequências de instruções, ele sugere a descoberta de idiotismos ou idiomatismos (*idioms*) pela decomposição. Uma sequência de instruções complexa é decomposta em uma sequência de instruções simples, e por este meio determina-se sequências de código ineficientes que podem ser substituídas por outras mais eficientes. Ou seja, esta técnica identifica restrições semânticas em uma sequência de instruções de tamanho arbitrário que a tornam equivalente a uma única instrução de propósito especial, denominada idiomatismo. A decomposição não é limitada a descobrir pares de instruções equivalentes; ela pode descobrir que uma instrução é equivalente a uma longa sequência de instruções. No pior caso, decomposição pode gastar menos tempo.

Sucintamente, a decomposição da descrição de instruções é feita da sequinte maneira: dada qualquer instrução, identificam-se todas as outras sequências de código que podem ser substituídas por aquela instrução. Este processo é repetido para cada instrução da máquina alvo, produzindo uma lista de todas as restrições de equivalência.

A maior contribuição desta técnica está no fato de que sequências de instruções podem ser estendidas para comprimentos arbitrários em uma tentativa de decompor uma instrução. A complexidade do processo de análise é exponencial, o grau de exponenciação depende do comprimento das sequências equivalentes que foram encontradas [Kessler, 1986]. Esta é uma propriedade importante, pois, quanto mais complexo for o conjunto de instruções mais tempo levará para analisar.

2.3.4 Abordagem de Fraser

Fraser [Fraser, 1989] descreve uma linguagem de programação para escrever geradores de código. A linguagem proposta por ele abrevia construções repetitivas, simplifica a codificação e torna os geradores de códigos menores e mais rápidos. Por exemplo, uma especificação para o VAX gasta 126 linhas, para o Motorola 68020 gasta 156 e para o MIPS R3000 são gastas 75 linhas. Sua técnica contrasta com os mais recentes métodos para geradores de código redirecionáveis, inclusive aqueles propostos pelo próprio autor: em primeiro lugar, os sistemas mais recentes aceitam descrições de máquina em uma representação não–procedural e produzem tabelas para serem interpretadas em tempo de compilação. Este sistema aceita uma representação compacta de um programa e emite um gerador de código *hard-coded*. A especificação do sistema possui um aspecto procedural, entretanto ela é menor que outras especificações. Em segundo lugar, os sistemas mais recentes utilizam-se de técnicas sofisticadas para gerar suas tabelas; em particular, este sistema usa um pré-processador, cuja operação é bem transparente. E, como nos analisadores sintáticos, qualquer um pode observar um analisador descendente recursivo e *"ver"* a gramática por trás, contudo é difícil *"ver"* qualquer padrão significativo em uma tabela LR. Em terceiro lugar, os sistemas mais recentes fiam–se em algoritmos de propósito geral, cuja aplicação abrange um universo maior que a geração de código. Por exemplo, os sistemas de Graham–Granvile [Aigrain et al., 1984], [Ganapathi and Fischer, 1985] fiam-se em analisadores LR. O sistema de Twig e Burs [Aho and Ganapathi, 1985] e [Graham and Pelegri-Llopart, 1988] fiam-se em avanços recentes em reconhecimento de padrão em árvores [Chase, 1987]. Sistemas baseados em otimizadores locais redirecionáveis [Davidson W. and Fraser W., 1984] baseiam-se em simulação simbólica. Em contraste, a técnica fundamental no sistema de Fraser é específica, correspondendo apenas à geração de código.

Programas são representados na linguagem para geração de código, principalmente através de regras de reescrita. Algumas regras reescrevem o código intermediário como um código simples em linguagem de montagem. Outras regras otimizam localmente o resultado. A linguagem de regras representa cada instrução da máquina alvo como uma instrução em linguagem de montagem, sob a forma de gabaritos *(templates)*. Por exemplo,

```
mov {b w l f d } y, z
{add sub mul div} {b w l f d}3 x, y, z
```

representa várias instruções VAX.

As regras de otimização, nesta abordagem, são escritas na mesma linguagem que as regras de geração de código, muito embora os idiomatismos sejam um pouco diferentes. Regras de geração de código casam código intermediário e produzem código objeto, enquanto as regras de otimizações casam códigos objeto e produzem códigos objeto melhorados.

Cada especificação é compilada em um programa C, denominado *rewrite*, que aceita DAGs (grafos acíclicos dirigidos) anotados com código intermediário, e gera, otimiza e emite código para a máquina alvo. Os geradores de código são usados com um analisador sintático para o *ANSI C*. Os compiladores resultantes emitem código similar ao *pcc1's* [Johnson, 1978], mas eles são executados duas vezes mais rápido.

É difícil avaliar a habilidade do método proposto, seu trabalho não apresenta exemplos para outras arquiteturas além do *VAX*, muito embora Fraser mencione o número de linhas gastas na especificação das arquiteturas do *MOTOROLA 68002* e do *MIPS R3000*.

2.3.5 Abordagem de Fraser e Wendt

Fraser e Wendt [Fraser W. and Wendt, 1986] e [Fraser W. and Wendt, 1988] propõem um compilador onde o gerador de código e o otimizador local dependente de máquina estão coesamente itegrados. Ambas as funções são efetuadas por um único sistema baseado na reescrita de regras, que casa padrões e substitui novos textos por eles. Esta organização torna o compilador mais simples, rápido e mais capaz de produzir um bom código.

O projeto, proposto por eles, se inicia com um otimizador local dirigido por regras, denominado *HOP* [Davidson W. and Fraser W., 1984], e o generaliza, pra também assumir as responsabilidades da geração de código. O compilador é redirecionável. As regras de geração de código são escritas manualmente, mas sua tarefa é simplificada pela ausência de análise de casos especiais. A necessidade de escrever estas regras é compensada pelo fato de a descrição da máquina [Davidson, 1981] necessária ser pequena o suficiente para tornar o método descrito competitivo com os outros métodos de compiladores redirecionáveis existentes [CATTELL 80].

Um conjunto de regras gera o código por meio da substituição do código intermediário por instruções da mquina alvo, representadas como transferência de registradores. Outro, o qual é geralmente criado automanticamente em tempo de compilação, otimiza este código tão logo ele seja produzido, substituindo instruções justapostas por uma única. Ainda outras regras traduzem as transferências de registradores otimizados para código em linguagem de montagem.

No sistema *HOP* quando um novo programa utiliza uma superposição de instruções que não foi vista quando as regras foram geradas em tempo de geração de compiladores *HOP* pode deixar de efetuar otimizações. Para corrigir esta dificuldade e tornar o sistema mais robusto, o sistema de Fraser e Wendt estende *HOP* em dois aspectos.Em primeiro lugar, integra *HOP* e *PO* [DAVIDSON 81] de tal forma que as regras em *HOP* sejam estendidas incrementalmente invocando *PO* para gerar regras para substituir ou rejeitar justaposições de instruções anteriormente não vistas. Estas regras são geradas em tempo de compilação,

mas nenhuma regra gerada anteriormente necessita ser produzida novamente. Portanto, o efeito é aproximadamente o mesmo da geração em tempo de geração de compiladores. Em segundo lugar, Fraser e Wendt estendem as regras de reescrita de tal forma que agora elas podem invocar rotinas *built-ins* para efetuar operações que não podem ser implementadas convenientemente com o reconhecimento de padrão e substituições. As regras de reescrita deste sistema se assemelham as regras de *HOP*.

Outro trabalho de Fraser e Wendt [FRASER 88], também nesta linha, utiliza-se dda notação para descrição da arquitetura de máquina delineada por [DAVIDSON 81].

2.3.6 Abordagem de Giegerich

Giegerich [Giegerich, 1983] apresenta um método para formalizar arquiteturas de máquinas, visando a derivação sistemática de otimizadores, onde a exatidão da otimização seja garantida. A derivação de otimizadores para máquinas específicas tem início a partir da descrição da máquina alvo. Uma descrição formal da semântica do conjunto de instruções define de forma precisa os conceitos característicos a nível de código de máquina, tais como: os modos de endereçamento e instruções, os efeitos colaterais, a superposição de registradores ou células de memória. A notação adotada para a descrição do conjunto de instruções é adaptada de ISP [Bell and Newell, 1971].

A descrição da máquina é analisada dentro de uma abordagem formal. A análise deriva predicados e funções que irão testar e inferir informações sobre o fluxo de dados. Estes predicados esclarecem dependências de máquina, dependências de fluxo de dados e propriedades dependentes do contexto. Os predicados dependentes de máquina nas instruções e modos de endereçamento são avaliados em tempo de geração de compiladores. Os predicados dependentes de programa devem ser avaliados em tempo de geração de código. Durante a geração de código, uma instrução é comparada com outra e substituída se for vantajoso. O otimizador aplica várias transformações independentes de máquina, explorando informações de fluxo de dados dependentes do programa. Estas transformações cobrem várias otimizações locais, incluindo eliminação de sub-expressões comuns e código redundante.

2.4 Quarta Família de Geradores de Código

Na quarta família de sistemas redirecionáveis de geração de código para arquiteturas CISCs tem-se o projeto do PQCC (*Production-Quality Compiler-Compiler*) [Cattell et al., 1979, Cattell, 1980, Leverett et al., 1980], o qual é um descendente do compilador Bliss-11 [Wulf et al., 1975].

2.4.1 Abordagem de Leverett

O projeto PQCC *Production-Quality Compiler-Compiler* [LEVERETT 80] é mais ambicioso que os trabalhos de Ganapathi e Glanville. O objetivo de *PQCC* é automatizar todas as fases envolvidas na construção de um compilador. O resultado prático deste objetivo é obter um sistema de geração de compiladores realmente automático. O projeto focalizou em duas áreas: otimização e na geração de código.

A seleção do código de *PQCC* é efetuada pelo gerador de gerador de código desenvolvido por Cattell [Cattell et al., 1979, Cattell, 1980]. A técnica de geração de código é similar a abordagem utilizada por Johnson [JOHNSON 78] no compilador C. Gabaritos (*templates*) são casados com a representação em forma de árvore do programa, entretanto, neste trabalho os gabaritos são gerados automanticamente a partir da descrição da máquina, o que não acontece com o método de Johnson.

Gabaritos são gerados usando métodos de pesquisa heurísticas, técnica oriunda de Inteligência Artificial. O uso de heurística não garante que a "melhor" seqüência de instrução seja encontrada, ou ainda, que alguma seqüência seja encontrada. Entretanto, na prática, heurística parece ser bem eficaz. As instruções na abordagem de Cattell são descritas como um conjunto de asserções que expressam a ação das instruções. Essas asserções são representadas como árvores, e uma notação semelhante a LISP é utilizada pelas árvores.Ks Cattell relata em sua dissertação que produz 2000 instruções por segundo no computador `DEC-10`.

Várias etapas inerentes ao gerador de código não são abordadas no gerador de código de Cattell. Tais etapas como: alocação e atribuição de registradores, atribuições a temporários e a memória são tratados por outras fase do compilador. Trabalhos realizados pelo grupo do projeto *PQCC* para a construção de geradores que deduzam estas informações dependentes de máquina da descrição da arquitetura [LEVERETT 80].

Concluindo, o projeto PQCC foi muito ambicioso: PQCC tentou incluir uma gama de detalhes muito minuciosa na linguagem para descrever a arquitetura da máquina alvo, de tal forma que compromissos entre espaço e tempo na otimização, seleção de código e alocação de registradores pudessem ser examinados. Devido ao fato de ter tentado incorporar muitos detalhes da máquina alvo, o sistema ficou difícil de ser utilizado.

2.5 Outros Sistemas

2.5.1 Abordagem de Warfield e Bauer

Warfield e Bauer [Warfield and Bauer, 1988] apresentam uma técnica que usa um Sistema Especialista com a missão de reconhecer que instruções podem ser otimizadas, a partir da descrição das instruções da máquina alvo. Uma ferramenta denominada "Meta-level Representation System" (MRS) [Russell, 1985] é utilizada na construção do sistema especialista, sendo sua característica principal a inclusão de um esquema de controle flexível utilizando raciocínio para frente, raciocínio para trás e uma técnica denominada resolução [Rich, 1983] para provar teoremas, além da habilidade de representar o conhecimento sobre ele mesmo (metalevel knowledge). MRS descreve uma teoria como um conjunto de fatos em sua própria biblioteca.

Neste sistema são definidas três teorias. A teoria de descrição de instruções, que descreve o que cada instrução faz, em termos das proposições entendidas pelo MRS. A teoria de modos de endereçamento, que descreve os modos de endereçamento e suas classes. E finalmente a teoria de otimização. Nesta teoria, utilizando o raciocínio para trás e um *peephole* de duas instruções, as regras tentam encontrar todas as otimizações possíveis usando certos critérios.

O otimizador de regras retorna uma lista de todas as otimizações possíveis e as condições que devem ser verdadeiras para que o otimizador funcione. Este resultado do Sistema Especialista é colocado na forma de regras e posto em uma teoria denominada Regras. Novas regras são criadas a partir daquelas que o otimizador retorna. O raciocínio para frente é então usado para otimizar o código objeto, utilizando estas regras. Cada linha do código objeto é lida como um fato na biblioteca, após ter sido codificada no formato definido por MRS. Após duas instruções terem sido declaradas, MRS pesquisa as regras automaticamente para encontrar uma que case as duas instruções. Se esta regra é encontrada, MRS instala a nova instrução otimizada na biblioteca. Esta nova instrução pode ser recuperada da biblioteca para ser usada na substituição de duas instruções originais no código objeto.

A linguagem de descrição proposta por Warfield e Bauer baseia-se na sintaxe de LISP. Ela provê, além de facilidades para descrever instruções e modos de endereçamento, e suas respectivas classes, um mecanismo para definir regras.

Para redirecionar o Sistema Especialista proposto para outra máquina, basta substituir a descrição das instruções na teoria de instruções e a descrição dos modos de endereçamento na teoria de modos de endereçamento.

2.6 A Linguagem de Davidson para Descrição de Arquiteturas

Para poder comparar as diferentes abordagens apresentadas até agora neste texto para as arquiteturas CISCs, vamos apresentar em mais detalhes a abordagem de Davidson.

A abordagem de Davidson [Davidson, 1981] compreende uma notação para a descrição da sintaxe e do efeito de cada instrução do conjunto de instruções das máquinas reais.

2.6.1 Conjunto de instruções do processador

Para trabalhar hoje com computadores, no nível de código de máquina, o pesquisador deve familiarizar-se com conjuntos de instruções de um grande número de novos processadores.

Esta tarefa, embora não seja um desafio intelectual, não é fácil. Apesar de não haver grandes diferenças nas noções básicas necessárias para o entendimento de conjuntos de instruções tradicionais, encontra-se dificuldade, ao estudar novos conjuntos de instruções, pela maneira como estes conceitos gerais são combinados; por exemplo, como o armazenamento do processador é arranjado, como as operações podem endereçar seus operandos, e quais operações são combinadas para formar uma simples instrução.

Na abordagem de Davidson, os efeitos das instruções são descritos usando uma notação semelhante a *ISP* [Bell and Newell, 1971]. Por exemplo, a instrução da máquina *DEC-10*

`add 3,loc` é expressa em *ISP* como: `r[3]` $\leftarrow$ `r[3]+m[loc]`

e significa que o conteúdo da posição de memória `loc` é somada ao registrador `3`.

Detalhes irrelevantes à descrição de máquina podem ser omitidos. Por exemplo, a instrução

`tst r1` do *PDP-11* é expressa em *ISP* como:

`NZ` $\leftarrow$ $r[1]$? 0;

onde, `NZ` representa o registrador de código de condição. Nesta instrução, o conteúdo do registrador "`1`" é comparado com zero, e o registrador de código de condição é atualizado de acordo com o resultado da operação. Os geradores de código não necessitam saber como o registrador de código de condição representa o resultado da comparação, assim sendo, a semântica do operador "`?`" não precisa ser especificada na descrição da máquina.

2.6.2 Descrição de Máquina

Na abordagem de Davidson, a arquitetura de máquina é especificada por meio de uma gramática para tradução dirigida por sintaxe. As produções nesta gramática são compostas de expressões e comandos simples, envolvendo os registradores e células de memória da máquina alvo. A linguagem proposta provê ainda facilidades para definir não-terminais, a sintaxe em linguagem de montagem para cada não-terminal e a sintaxe para a correspondente transferência entre registradores e modos de endereçamento. A descrição de máquina é logicamente dividida em duas partes. A primeira parte descreve os modos de endereçamento da máquina, e a segunda parte descreve as instruções. Esta divisão proporciona um modo natural de descrever arquiteturas de máquinas. Os modos de endereçamento são descritos sem considerar as instruções que os usam. As operações de máquina e os modos de endereçamento são combinados para descrever o conjunto de instruções da máquina. A vantagem de estruturar o conhecimento sobre uma arquitetura desta forma é produzir, como resultado, uma descrição de máquina concisa e legível.

No trabalho de Davidson, o reconhecedor e o tradutor mencionados na seção 2.3 são obtidos transformando a descrição de máquina em uma gramática para o gerador de analisadores léxico *LEX* [Aho et al., 1986]. A partir da entrada, *LEX* gera subrotinas que implementam o reconhecedor e o tradutor. Como *LEX* reconhece somente expressões regulares, a máquina alvo deve ser descrita usando expressões regulares. Muitas máquinas, contudo, possuem em suas instruções, componentes que são sensíveis ao contexto. Por exemplo, a instrução *add* do computador *DEC-10* pode ser descrita como:

$$\texttt{RG} \leftarrow \texttt{RG + M}$$

onde `RG` representa um registrador, e `M` representa uma posição de memória. Quando um não-terminal aparece duas vezes em uma produção, os *strings* casados pelos padrões devem ser idênticos. Neste exemplo, as duas instâncias de `RG` devem casar o mesmo *string*.

Às vezes não se pode considerar a verificação da sensibilidade do contexto efetuada pelo reconhecedor. Isto pode ser feito definindo várias instâncias de um mesmo padrão, cada uma delas com nomes diferentes. Por exemplo, a descrição da instrução de movimentação de registradores do Cyber CDC é:

$$\text{x[RN1]} \leftarrow \text{x[RN2];}$$

Muito embora as instâncias RN1 e RN2 representem o mesmo padrão, elas não precisam casar *strings* idênticos.

Considerando que expressões regulares não são suficientemente poderosas para descrever arquiteturas de máquina, Davidson desenvolveu um trabalho [Davidson, 1985], onde ele utiliza-se de uma descrição de máquina baseada na teoria de linguagens livres do contexto, e de um gerador de analisador sintático como YACC, para processar a especificação de uma linguagem de descrição. O método utilizado soluciona os problemas levantados no parágrafo anterior.

Em sua abordagem, uma descrição de máquina é composta de uma gramática e ações que descrevem a sintaxe e a semântica do conjunto de instruções da arquitetura alvo. Assim, escrever uma descrição de máquina é semelhante a escrever um analisador sintático para uma linguagem de programação.

2.6.3 Exemplo de uma descrição de máquina

Esta seção apresenta partes da descrição de máquina para o *PDP-11*.

```
RN      [0-7]+
XDENT (((“_” | “L”)[A-Za-z0-9_]+) | (-?[0-9]+)
IDENT  XDENT(“ ” [-+] “ ” XDENT)*
LABEL  “L” [L0-9]+
```

Esta parte da descrição define expressões regulares utilizadas ao longo da descrição dos modos de endereçamento. A primeira definição descreve um número de registrador: uma sequência de um ou mais dígitos entre 0 e 7. A segunda definição descreve um componente de um identificador. `IDENT` define um identificador: um `XDENT` seguido de zero ou mais ocorrências dos operadores “+” ou “-” seguindo por um `XDENT`. A última definição descreve rótulos. A seguir são definidos os modos de endereçamento.

```
RG      := r[RN]                    := rRN
LB      := LABEL                    := LABEL
        := 0                        := $0
        := 1                        := $1
ID      := IDENT                    := $IDENT

WORD := m[IDENT]                    := IDENT
        := m[r[RN] + IDENT]         := IDENT(rRN)
        := m[r[RN]++]               := (rRN)+
        := m[--r[RN]]               := -(rRN)
        := m[r[RN]]                 := (rRN)
```

```
        := m[m[r[RN]++]]            := *(rRN)+
        := m[m[--r[RN]]]            := *-(rRN)
        := m[m[r[RN] + IDENT]]      := *IDENT(rRN)
        := m[m[IDENT]]              := *IDENT
        := m[m[r[RN]]]              := *(rRN)

BT      := b[IDENT]                 := IDENT
        := b[r[RN] + IDENT]         := IDENT(rRN)
        := b[r[RN]++]               := (rRN)+
        := b[--r[RN]]               := -(rRN)
        := b[r[RN]]                 := (rRN)
        := b[m[r[RN]++]]            := *(rRN)+
        := b[m[--r[RN]]]            := *-(rRN)
        := b[m[r[RN] + IDENT]]      := *IDENT(rRN)
        := b[m[IDENT]]              := *IDENT
        := b[m[r[RN]]]              := *(rRN)

REL     := ==                       := eq
        := =                        := ne
        := ≥                        := ge
        := ≤                        := le
        := <                        := lt
        := >                        := gt

SO      := <<
NZ      := NZ
PC      := PC
```

Cada linha da definição acima possui três campos: o *token* retornado pelo reconhecedor, se o padrão para a transferência de registradores casa, a sintaxe na forma *ISP* para o padrão, e a sintaxe em linguagem de montagem para o padrão. Por exemplo, a primeira linha acima define um registrador *r* seguido pelo número do registrador entre colchetes. A sintaxe em linguagem de montagem correspondente é *r* seguido pelo número do registrador. Quando um registrador é reconhecido, o *token RG* é retornado. Se o primeiro campo for vazio, o *token* retornado é o *string* casado, como aparece na terceira linha da definição. O último campo, correspondente à sintaxe em linguagem de montagem para o padrão, também é opcional, como mostra a definição de `SO` (operador *shift*). A parte final da definição de endereço agrupa os *tokens* e os caracteres simples em classes. Este agrupamento permite uma definição simples e concisa das instruções, pela combinação dos operadores e modos de endereçamento semelhantes.

```
RG1     := RG
```

RG2 := RG
DSTW := RG | WORD
SRCW := RG | ID | WORD | 0 | 1

A sintaxe para a definição de instruções é:

instruction expression := *instruction definition*
instruction definition := *action*
| { [*test condition* :] *action*
[*test condition* :] *action*
...
}

Colchetes representam um campo opcional, e "|" separa as alternativas. Elipses ("...") representam repetição indefinida de itens; *instruction expression* é a representação em *ISP* da instrução; *action* é executada se o *test condition* a ela associada é verdadeiro; *test condition* é avaliado na ordem em que aparece. Uma *instruction expression* casa com o *string* de entrada se ela efetua todas as transferências de registradores requisitadas. Seguem dois exemplos de definição de instruções para o *PDP-11*, instruções *cmp*, *asr* e *ash*:

```
NZ ← DSTW ? SRCW; := cmp DSTW,SRCW
```

No exemplo abaixo, o *test condition* pode invocar procedimentos fornecidos pelo sistema ou pelo usuário. O *test condition* permite que *instruction expression* identifique duas instruções. Se a rotação (*shift*) é "-1", então a instrução *asr* casa, senão a instrução mais geral *ash* casa. Isto é possível através do procedimento *strcmp*. Este procedimento compara dois *strings* e retorna verdadeiro, se eles são iguais, caso contrário, retorna falso.

```
RG ← RG SO SRCW; NZ ← RG SO SRCW ? 0; := {
        !strcmp(SRCW, "-1") :  asr RG
                ash SRCW,RG }
```

2.7 Comparação da Abordagem de Davidson com Outros Trabalhos

Os diversos trabalhos apresentados na seção 2 utilizam-se da descrição de máquina para obter a independência das arquiteturas de máquina. Uma vez que cada método depende fortemente destas descrições, vale a pena comparar e contrastar as várias abordagens de

descrição. A instrução "`add`" do *PDP-11* é utilizada como base para a comparação. Entretanto, existem algumas abordagens que não fornecem informações suficientes para elaborar um exemplo nesta máquina, nestes casos, as arquiteturas a que a instrução "`add`" se refere são explicitamente indicadas antes do exemplo.

2.7.1 Primeira Família de Geradores de Código

Abordagem de Glanville
Na abordagem de Glanville uma instrução *add* é descrita da seguinte maneira:

r.1 ::= (+r.1 r.2)	"add r.2,r.1";
r.1 ::= (+k.1 r.1)	"add $k.1,r.1";
r.1 ::= (+ ↑ k.1 r.1)	"add *k.1,r.1";
λ ::= (:= k.1 + ↑ k.1 r.1)	"add r.1, *k.1";
r.2 ::= (+ ↑ + k.1 r.1 r.2)	"add k.1(r.1), r.2";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 r.2)	"add r.2, k.1(r.1)";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 ↑ + k.2 r.2)	"add k.2(r.2), k.1(r.1)";
λ ::= (:= + k.1 r.1 + k.2 ↑ + k.1 r.1)	"add $k.2, k.1(r.1)";

`r.1` e `r.2` representam registradores; `k.1` e `k.2` representam constantes. ↑ representa um operador unário que recupera o valor da posição de memória endereçada por seu operando. A instrução na linguagem de montagem (*assembler*), que aparece à direita da produção, corresponde à instrução a ser emitida se a produção casa com a entrada.

Abordagem de Henry
Na abordagem de Henry, uma instrução *add* é descrita da seguinte maneira:

```
Plus reg src'1
reg
cg("add2 $src'1,$reg", "$reg")
cost(1)
```

`Plus reg src'1` representa o padrão na forma prefixada; `reg` representa a substituição; `cg("add2 $src'1,$reg" "$reg")` representa a ação e `cost(1)` representa o custo. O predicado para esta instrução não foi especificado, portanto assuma-o verdadeiro.

Abordagem de Costa
Na abordagem de Costa, a instrução *add* pode ser descrita da seguinte maneira:

```
regra modoI    : modoI := k;                 FazDesloc k;
regra modoR    : modoR:= r;                  FazBase r;
regra modo1    : modo1 := + modoI modoR;     SomaDescrs modoI modoR;
regra modo2_1 : modo2 := ↑ modo1;            CopiaDescr modo1; IncNivel modo2;
regra modo2_2 : modo2 := modoI;              CopiaDescr modoI;
regra modo2_3 : modo2 := modoR;              CopiaDescr modoR;
regra modo3_1 : modo3 := ↑ modo2;            CopiaDescr modo2; IncNivel modo3;
regra modo3_2 : modo3 := modo2;              CopiaDescr modo2;
```

```
regra addword1 : r.1                 :=                 word + modo3 r.1;
                 Emite 'add'         modo3 ',' r.1;     custo 2;
regra addword2 : r.1                 :=                 word + r.1 modo3;
                 Emite 'add'         modo3 ',' r.1      custo 2;
regra addword3 : lambda              :=                 word st modo3 word+↑ modo3 modo2;
                 IncNivel modo3;
                 Emite 'add word'    modo2 ',' modo3;   custo 2;
regra addword3 : lambda              :=                 word st modo3 word+modo2 ↑ modo3;
                 IncNivel modo3;
                 Emite 'add word'    modo2 ',' modo3;   custo 2;
```

`FazBase, FazDesloc, CopiaDescr, IncNivel e SomaDescrs` são operadores semânticos disponíveis na linguagem descritiva.

`modo1, modo2, modo3, modoI e modoR` são não-terminais definidos para agrupar os diversos modos de endereçamento.

`r.1` representa registrador;

`k` representa as constantes.

↑ representa um operador unário que recupera o valor da posição de memória endereçada por seu operando.

2.7.2 Segunda Família de Geradores de Código

Abordagem de Ganapathi
Na abordagem de Ganapathi uma instrução *add* é descrita da seguinte maneira:

Word↑ → + Word↑a Word↑r Istemp(↓r) EMIT (↓'add' ↓a ↓r)
→ + Word↑a Word↑b GETTEMP(↓'word' ↑r)
EMIT(↓'mov' ↓b ↓r)
EMIT(↓'add' ↓a ↓r)

Todas as produções são da forma "*LHS* → *RHS*". O *LHS* é um único não-terminal, normalmente contendo atributos sintetizados. O *RHS* contêm terminais com atributos sintetizados, não-terminais com atributos sintetizados, predicados para retirada de ambiguidade (sublinhado) com atributos herdados e símbolos de ação (letras maiúsculas), possuindo atributos sintetizados e herdados. ↑ representa um atributo sintetizado que transmite informação acima na árvore; ↓ representa um atributo herdado, ele passa informação para baixo na árvore sintática. As variáveis `a, b` e `r` são variáveis de atributo. `Word` representa o modo de endereçamento baseado em palavra, disponível no computador *PDP-11*. `Istemp` é um predicado responsável pela retirada de ambiguidade, ele determina quando a produção corrente pode ser aplicada. `GETTMP` requisita um temporário, e `EMIT` gera a instrução em linguagem de montagem, se a produção casa com a entrada.

2.7.3 Terceira Família de Geradores de Código

Abordagem de Davidson

Na abordagem de Davidson [Davidson, 1981], uma instrução *add* é descrita da seguinte maneira:

```
DSTW ← DSTW + SRCW; 0NZ ← DSTW + SRCW ? 0; := add SRCW, DSTW
```

`DSTW` e `SRCW` representam os modos de endereçamento com palavras, disponíveis no computador *PDP-11*. `NZ` representa o registrador de código de condição. A instrução em linguagem de montagem que aparece à direita de "`:=`" é emitida, se a expressão representando a instrução se casa com a entrada.

Abordagem de Robert R. Kessler

Na abordagem de Kessler, uma instrução *add* para a arquitetura *MC68000* é descrita da seguinte maneira:

```
((add.1 EA-All Dn)
(Make-Flags (setf Dn (+ Dn EA-All)) Dn * * * * *)
(+ 8 (EA-Time EA-All 4)) % Time & Space depend
(long EA-An)) % on Adrressing mod0e
```

`EA-All` representa todos os modos de endereçamento para a arquitetura *MC68000*, `EA` significa o modo de endereçamento efetivo. `EA-An` representa `EA` sem `An`. `Dn` representa o registrador de dados. Os "$*$" representam os códigos de condição, `X, N, Z, V e C` respectivamente. `Make-flags` é uma macro que, a partir de uma expressão e uma lista de código de condições, retorna uma expressão com todos os valores definidos.

Abordagem de Peter B. Kessler

Na abordagem de Peter B. Kessler, uma instrução de adição de palavra longa composta de três operandos, `ADDL3`, para a arquitetura *VAX-11*, é descrita da seguinte maneira:

(in-order
(←a-dest (+ a-src_2 a-src_1))
(any-order
(← cc-n (< a-dest 0))
(← cc-z (= a-dest 0))
(← cc-v (overflow a-dest))
(← cc-z(carry a-dest))))

Os operadores `in-order` e `any-order` representam a sequência da computação. "`cc's`" representam os códigos de condição que a arquitetura do *VAX-11* atualiza durante a execução da maioria das instruções aritméticas. A notação prefixada parentetizada é usada para mostrar como a árvore define a computação.

Abordagem de Fraser

Na abordagem de Fraser, uma instrução *add* é descrita da seguinte maneira:

.==“ADDI” .=%f%t3 %x,%y,%z
f = “add”
t = “1”
xm = “r%n” ym = “r%n” zm = “r%c”
yn = K0
xn = K1

A linguagem de regras possui dois operadores básicos: “==” testa, e “=” assinala. “`%f`” representa um operador binário, por exemplo, `add, sub, etc.`. “`%t`” representa o tipo do sufixo `(b, w, f, 1, d)`. “`%y`” e “`%z`” são os operadores gabaritos. “`yn`” e “`xn`” representam apontadores para os nodos que calculam os endereços dos operandos. As regras podem ser abreviadas pela substituição das constantes *strings* pelos seus *placeholders*, de tal forma que a regra acima, definindo a instrução *add*, poderia ser expressa da seguinte maneira:

.==“ADDI” .= “add13” “r%n,r%n,r%c”
yn = K0
xn = K1

Para interpretar um gabarito em linguagem de montagem, ou seja, para gerar uma saída, *placeholders* são substituídos com os valores dos campos correspondentes. Por exemplo, se *f* representa *add* e *t* representa *1* então o gabarito *%f%t3* representa *add13*.

Abordagem de Giegerich

Na abordagem de Giegerich, uma instrução *add* para a arquitetura *MC68000* é descrita da seguinte maneira:

ADD_1: `Dreg := Dreg + all, Cy := carry(Dreg + all),`
`Z := (Dreg + all = 0), N := (Dreg + all < 0) cost2.2;`

`Dreg` especifica o modo de endereçamento, neste caso, o registrador de dados; `all` representa a classe do operando que pode ser: `Dreg, Areg, disp, postinc, im32,...`, onde, `Areg` é o registrador de endereço, `disp` é o modo de endereçamento de palavra-dupla (base

+ deslocamento), `postinc` é o modo de endereçamento pós-incremento, e `im32` representa constantes. `Cy, Z e N` representam os códigos de condição, `Cy` define o ``carry'', `X e V` foram omitidos nesta descrição.

2.7.4 Quarta Família de Geradores de Código

Abordagem de Cattell

Na abordagem de Cattell instruções são sintaticamente representadas sob a forma de assertivas, escritas em notação parentetizada, no estilo *LISP*. A descrição da instrução *add* é:

(; (← $1:DST (+ $1:DST $2:SRC)) (← %N (LSS (+ $1:DST $2:SRC) 0))
(← %Z (EQL (+ $1:DST $2:SRC) 0)) :: (EMIT [ADD 2 1 1]6 $2 $1)

As instruções são representadas como produções. O lado esquerdo é formado pelas asserções de entrada e saída que definem as ações das instruções. Os componentes do lado direito, a partir de `EMIT`, especificam o custo espaço/tempo, formato, mnemônico e uma lista de valores dos campos das instruções. `DST` e `SRC` representam os modos de endereçamento com palavras disponíveis no computador *PDP-11*. Eles são descritos em uma parte separada da descrição. `$1` é usado para indicar que os registradores de destino e do operando devem ser o mesmo registrador. Ambos "`$1 e $2`" são usados para referências posteriores na determinação dos campos reais, isto é, número dos registradores. "2" indica operação de dois operandos. Os dois "1's" representam o custo relativo ao espaço/tempo respectivamente.

2.7.5 Outros Sistemas

Abordagem de Warfield

Na abordagem de Warfield uma instrução *add* para o *VAX* é descrita da seguinte maneira:

```
(instruction addb3)
(operation addb3 addition)
(left addb3 operand3 all-modes)
(right1 addb3 operand2 all-modes)
(right2 addb3 operand1 all-modes)
(set addb3 NZ)
(size addb3 1)
```

```
(space addb3 3)
(time addb3 3)
```

`all-modes` representa todos os modos de endereçamento, os três operandos desta instrução podem ser usados com qualquer modo de endereçamento. A proposição `set` indica que os bits do código de condição "N" e "Z" devem ser atualizados após a operação de adição. A proposição `size` representa o tamanho dos operandos, nesta operação "1" diz que `addb3` é uma instrução *byte*. As constantes especificadas nas proposições `space` e `time` de uma instrução são arbitrárias, mas devem ser consistentes com o restante das outras instruções presentes na descrição.

2.8 Avaliação das Abordagens para Arquiteturas CISCs

A descrição de máquina proposta por Davidson [Davidson, 1981] é, de certa forma, mais fácil de ler que as outras apresentadas, porque ela utiliza a notação infixada ao invés da notação prefixada. A descrição de Ganapathi requer que o usuário implemente os predicados de retirada de ambiguidade e as ações para cada máquina. Enquanto isto torna o método flexível, no que tange às otimizações independentes da arquitetura da máquina alvo, complica a elaboração da descrição da mesma. A descrição proposta por Glanville é substancialmente mais longa que as outras apresentadas, devido aos modos de endereçamento não serem fatorados na definição das instruções. Este problema é resolvido na abordagem apresentada por Costa [Costa, 1990]. A vantagem da técnica delineada por Costa em relação a Glanville reside, exatamente, na fatoração da gramática descritiva. Reduzindo a gramática, também diminui o tamanho das tabelas geradas automaticamente pelo sistema. Contudo, o tempo gasto na geração de código pode aumentar, devido às novas reduções introduzidas pela fatoração, constituindo, assim, uma desvantagem da abordagem de Costa. Ambas as descrições, de Glanville e Ganapathi, são incompletas, no sentido de que o teste e a atualização dos códigos de condição não são descritos. A mesma observação é válida para o método de Costa, se o objetivo é otimizar código. No caso da abordagem de Glanville, isto significa que nenhum código pode ser gerado ao efetuar uma adição, unicamente pelo efeito colateral de alterar o código de condição. Ganapathi possui um mecanismo separado que vasculha por instruções que são utilizadas somente para ajustar o código de condição. Caso ele ajuste o código de condição exatamente como na instrução anterior, a instrução é suprimida. Como a descrição da instrução *add* não especifica explicitamente que a instrução ajusta o código de condição, supõe-se que esta informação deva vir de alguma outra fonte. A descrição de máquina de Cattell é mais complexa que as outras apresentadas. A definição de máquina através de axiomas não é prática para uma variedade de instruções de máquina. Sua abordagem, em termos de complexidade, compara-se com a abordagem de Ganapathi, muito embora os métodos utilizados por ambos sejam diferentes.

As demais técnicas apresentadas na seção 2 [Kessler, 1984], [Kessler, 1986], [Henry, 1987],

[Henry and Damron, 1989b], [Henry and Damron, 1989a], [Fraser, 1989] não oferecem informações suficientes para que se possa avaliá-las. Contudo, de todas estas abordagens, a que parece mais simples é aquela proposta por R. Kessler [Kessler, 1984].

A descrição de Giegerich é baseada na formalização da semântica do conjunto de instruções. Ela compreende um modelo abstrato de conjunto de instruções de uma maneira geral, uma notação para a descrição de máquina real, baseada em *ISP* [Bell and Newell, 1971], além da semântica para tal descrição, que é apresentada em termos do modelo abstrato. Este enfoque resulta em uma descrição de instruções simples, entretanto a forma utilizada para descrever os modos de endereçamento é complicada e trabalhosa. A abordagem utilizada por Warfield e Bauer para descrever arquiteturas de máquinas é compacta, preenche todos os requisitos necessários em uma linguagem de descrição, inclusive mecanismos para definir regras. Entretanto, depende de uma ferramenta, *MRS* [Russell, 1985], que está disponível somente em três máquinas: *DEC-20* rodando *MacLISP*, *VAX* rodando *FranzLISP* sobre o *UNIX* de Berkeley e "*Symbolics LISP machine LM-2/3600*", o que torna o seu uso inviável.

A técnica de descrição utilizada por Davidson provê um método simples, muito embora robusto, para descrever instruções de máquina. Detalhes irrelevantes da arquitetura da máquina podem ser omitidos, enquanto que aqueles complicados, mas necessários, podem ser facilmente descritos. Todas as descrições de arquiteturas apresentadas neste capítulo, através de exemplos, possuem algumas características em comum. Em primeiro lugar, elas definem modos de endereçamento e instruções das arquiteturas de máquina, algumas de forma mais complicada, outras mais simples, do ponto de vista de legibilidade. Em segundo lugar, quase todas utilizam a notação *ISP*, ou adaptação dela [Bell and Newell, 1971], como base nas descrições. Acreditamos que isto seja porque *ISP* provê um mecanismo preciso e sem ambiguidades para especificar arquiteturas de máquinas. Em terceiro lugar, quase todas processam algum tipo de gramática. Glanville e Costa usam gramáticas livres do contexto, Ganapathi utiliza gramáticas de atributos, Davidson usa expressão regular. Outros utilizam-se de regras, por exemplo, Fraser e Wendt, Henry, Warfield e Bauer. Finalmente, quase todas fatoram de alguma forma os diferentes modos de endereçamento das arquiteturas em questão.

Considerando que uma linguagem de descrição de máquina serve como veículo para especificar o comportamento de uma máquina alvo, é importante que a mesma possua mecanismos para definir o máximo de informações possível sobre uma dada arquitetura. Dependendo da aplicabilidade da descrição de máquina, é fundamental que ela seja capaz de especificar outras características da máquina, além de seu conjunto de instruções e modos de endereçamento. Por exemplo, a inclusão de facilidade para definir custo relativo a tempo e espaço é característica importante, quando o objetivo é otimizar o código gerado. Outras informações, como por exemplo, mecanismos para atualizar o código de condição utilizado nas operações lógico-aritméticas, tornam as notações mais poderosas. A aplicabilidade dos métodos apresentados não varia muito, a maioria deles visa as arquiteturas *PDP-11, VAX-11, MOTOROLA 68000*. Por exemplo, Glanville discute a implementação de gera-

dores de código para as máquinas *PDP-11* e *IBM-370*. Ganapathi produziu geradores de código para as arquiteturas *VAX-11, PDP-11* e *Intel 8086/8087*. Cattell discute a implementação de geradores de código para *IBM-360, PDP-10, PDP-11, Intel 8080, Motorola 6800* e *PDP-8*. Costa apresenta a implementação de um sistema de produção de geradores de código para o processador *Intel 8088*; sua técnica também é aplicável aos processadores *IBM-360* e *PDP-11*. Robert Kessler discute a implementação de *PEEP*, um otimizador local, atualmente integrado no compilador *LISP* na arquitetura *Motorola 68000*. O trabalho de Giegerich é direcionado para as arquiteturas *MC86000* e *8086*, mas a maioria das otimizações são também relevantes para máquinas de grande porte. Peter Kessler descreve uma técnica para analisar descrições de máquina visando o uso de instruções de propósito especial. Sua ferramenta analisa duas arquiteturas, *VAX-11* e *MC68000*. A linguagem para escrever geradores de código proposta por Fraser [Fraser, 1989] se aplica nas arquiteturas *VAX* e *MC68020*. O compilador proposto por Fraser e Wendt, no qual o gerador de código e o otimizador estão coesamente integrados, gera código para a arquitetura *VAX*. A máquina alvo utilizada por Warfield e Bauer também é o *VAX*. O método proposto por Davidson [Davidson, 1981], *PO*, é aplicável nas arquiteturas: *CDC Cyber 175, DEC-10, PDP-11* e no microprocessador *8080* da Intel. Estas máquinas representam arquiteturas bem diferentes. O *CDC Cyber 175* e o *DEC-10* são dois tipos distintos de computadores de grande porte. O *PDP-11* é um minicomputador de 16 *bits*, enquanto o *8080* é um microprocessador de 8 *bits*.

Nas abordagens apresentadas, nenhuma pesquisa foi realizada para explorar o paralelismo de *CPU*, a não ser [Davidson, 1981]. Gerar código para arquiteturas não convencionais, como por exemplo processadores vetoriais e máquinas possuindo paralelismo, é difícil. Estudos realizados por Davidson sugerem que a inclusão em *PO* de informações adicionais sobre o processador através da descrição da máquina pode ser possível gerar código para o processador de maneira convencional. *PO* aplicaria transformações no código, produzindo um código que se aproveita das arquiteturas não convencionais. Por exemplo, nas máquinas com múltiplos processadores aritméticos, poderia ser gerado código que se utiliza somente de um processador para cada instrução. *PO* pode substituir pares de instruções que utilizam processadores diferentes por uma única instrução que usa ambos.

Nas arquitetura com múltiplas unidades funcionais, por exemplo Cyber CDC, é conveniente distanciar o uso dos resultados de operações de suas próprias computações. Isto permite ao processador prosseguir, não tendo que esperar por resultados que ainda não estão disponíveis. Para este caso, PO poderia efetuar o escalonamento de instruções reordenando o código. Enquanto elimina código redundante PO poderia construir listas contendo passos intermediários das computações. Aplicando um algoritmo de escalonamento em tais listas PO poderia rearranjar os passos da computação para evitar a espera do processador.

A tabela a seguir sintetiza os pontos mais importantes dos métodos apresentados para arquiteturas CISCs.

	descrição máquina	método usado	redirecionamento
GLANVILLE	muito extensa incompleta	gramáticas livres do contexto LR	difícil avaliar dependência das linguagens:fonte e descritiva
GANAPATHI	longa, difícil e incompleta	gramáticas de atributos	difícil avaliar arquiteturas semelhantes
CATTELL	mais complexa, equipara-se a Ganapathi em complexidade	definição na forma de asserções de entrada e saída associada a cada instr.	distribuição de etapas em outras partes do compilador
COSTA	muito longa	gramáticas livres do contexto	dependência de máquina da repres. intermediária embutida no frontend
DAVIDSON	fácil de ler, flexível	expressão regular gramáticas livre contexto-lex,yacc	mais fácil
WARFIELD	compacta, mas depende de MRS	regras	difícil avaliar
GIEGERICH	simples porem forma utilizada para descrever modos de endereçamento complicada, trabalhosa	adaptação de ISP	restringe somente a MC86000, 8086
HENRY	razoável	enumeração de cinco tuplas PPRACS ou regras	difícil avaliar

3 Geradores de Geradores de Código para Arquiteturas RISCs

Para especificar as características dos processadores RISC, linguagens de descrição de arquiteturas com as propriedades descritas até agora não são suficientes. Uma linguagem para estas máquinas deve possuir mecanismos que lhe permitam tratar o escalonamento de instruções de forma satisfatória.

3.1 A LDA para Arquiteturas Superescalares

Considerando que uma linguagem de descrição de máquina serve como veículo para especificar o comportamento de uma máquina alvo, é importante que a mesma possua mecanismos para definir o máximo de informações possível sobre uma dada arquitetura. Dependendo da aplicabilidade da descrição de máquina, é fundamental que ela seja capaz de especificar outras características da máquina, além de seu conjunto de instruções e modos de endereçamento. Por exemplo, a inclusão de facilidade para definir custo relativo a tempo e espaço é característica importante, quando o objetivo é otimizar o código gerado. O conhe-

cimento do custo é necessário para dirigir o processo de otimização. Outras informações, como por exemplo, mecanismos para atualizar o código de condição utilizado nas operações lógico-aritméticas, tornam as notações mais poderosas.

A LDA para a especificação de arquiteturas de máquinas superescalares que apresentamos neste trabalho leva em consideração as questões levantadas na Seção **??** e incorpora, de forma que consideramos satisfatória, mecanismos para definição dos modos de endereçamento, facilidades para descrever as unidades funcionais, os estágios do *pipeline* e outras propriedades para o escalonamento; permite definição completa da semântica das instruções; incorpora informações sobre a geração e otimização de código; identifica as classes de arquiteturas que podem ser descritas e é concisa e possui alto poder de expressão. Ela é uma adaptação da linguagem proposta por Bradlee [Bradlee et al., 1991], incorpora as características básicas desta linguagem, mas inclui novas facilidades, amplia seu leque de abrangência e a torna mais robusta. Ela é capaz de especificar os pontos abaixo relacionados necessários para descrever arquiteturas superescalares.

1. Características gerais das arquiteturas:
   - Registradores.
   - Estágios de *pipeline*.
   - Unidades funcionais.
   - Memória.
2. Modelo da máquina virtual.
   - Definição do uso dos registradores.
   - Passagem de parâmetros.
3. Lista de Instruções:
   - Instruções padrão para cada instrução de máquina deve ser especificado:
     - Mnemônico da instrução.
     - Restrições de tipo dos operandos.
     - Semântica da instrução.
     - Recursos de *pipeline* necessários.
     - Outras propriedades de escalonamento, como por exemplo. latência, custo e número de *slots* associado às operações.
   - Instruções especiais.
4. Detalhes específicos de arquitetura:
   - Transformações necessárias para auxiliar no mapeamento da linguagem intermediária com o conjunto de instruções da máquina alvo.
   - Especificações de latência especiais.

3.1.1 Estrutura da LDA

Para maior clareza, a LDA é composta de três seções: seção de declarações, seção com as características da máquina alvo e seção de instruções. A especificação das declarações deve necessariamente iniciar com a palavra "**declarations** {" e finalizar com "}" . Nesta seção são declarados os registradores, os recursos da máquina, as constantes, o tamanho da memória disponível entre outras informações. A seção **vm** especifica as características da máquina alvo. Esta seção inicializa com a palavra "**virtual_machine**" seguida do símbolo "{" e finaliza com o símbolo "}" . 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. Esta seção inicializa com a palavra chave "**instructions**" seguida do símbolo "{" e finaliza com o símbolo "}" .

3.1.2 Seção de Declarações

Esta seção consiste em uma lista de definições que especificam as características da arquitetura alvo. Para cada definição existe uma palavra iniciada com "%" que a identifica. A Figura 2 mostra um trecho da descrição da seção de declarações. São declaradas nesta seção:

- Definição de registradores: conjuntos de registradores disjuntos para operações de ponto flutuante e inteiros ou um conjunto de registradores compartilhado entre estes dois tipos de operações.
- Sobreposição de registradores: através da diretiva %equiv, o projetista do compilador descreve a sobreposição do conjunto de registradores. Esta facilidade pode ser usada em arquiteturas que possuem um conjunto de registradores compartilhado para as operações com inteiros e ponto-flutuante.
- Recursos da máquina: declara os recursos computacionais da máquina, por exemplo, estágios da *pipeline*.
- Constantes: define um nome para representar um valor dentro de um intervalo.
- Memória disponível: define um arranjo contendo as localizações de memória.
- Definição de Rótulos: define um nome para representar um rótulo com endereço relativo.
- Declaração de Relógios: define um nome para ser um relógio. Este relógio acompanha o processamento de sub-operações durante o escalonamento temporal[4] em arquitetu-

[4]Escalonamento temporal é o processo de acompanhar a colocação dos resultados das sub-operações em registradores temporários.

ras que possuem *pipeline de avanço explícito*[5]

- Definição de Classes: define classe para ser o conjunto de elementos previamente definidos.
- Associação de Instruções a Relógios: define um nome que representa uma instrução como um elemento pertencente a um conjunto classe.

```
declarations
{
  %reg r[0:31] (char|int|short|pointer);
  %reg i[0:7]   (char|int|short|pointer);
   %reg f[0:31] (float|int);
    %reg d[0:15] (double);
    %clock clk-cc;
     %reg cc(int; clk-cc) +temporal;

    %equiv i[0]     r[24];

     /* Integer Unit Stages  (CY7C601) */
          %resource IF;   /* instruction fetch */
          %resource ID;   /* decode */

      %memory m[0:4294967295];

      %def uns-const[65536:2147483647]  +relocatable;

       %label rel-lab[-4194304:4194303]  +relative
  }
```

Figura 2: Trecho da Seção de Declarações para uma dada Arquitetura

Informações coletadas das diretivas da seção de declarações são colocadas em uma matriz e posteriormente utilizadas nas rotinas para construir o DAG de código[6], para gerar código, para efetuar reconhecimento de padrões, para alocar registradores, para fazer transformações e para escalonar instruções.

3.1.3 Seção de Definição da Máquina Alvo

A seção (`vm`) descreve o modelo de execução da máquina à qual o código gerado deve corresponder. Seus objetivos são, basicamente: especificar as convenções das chamadas

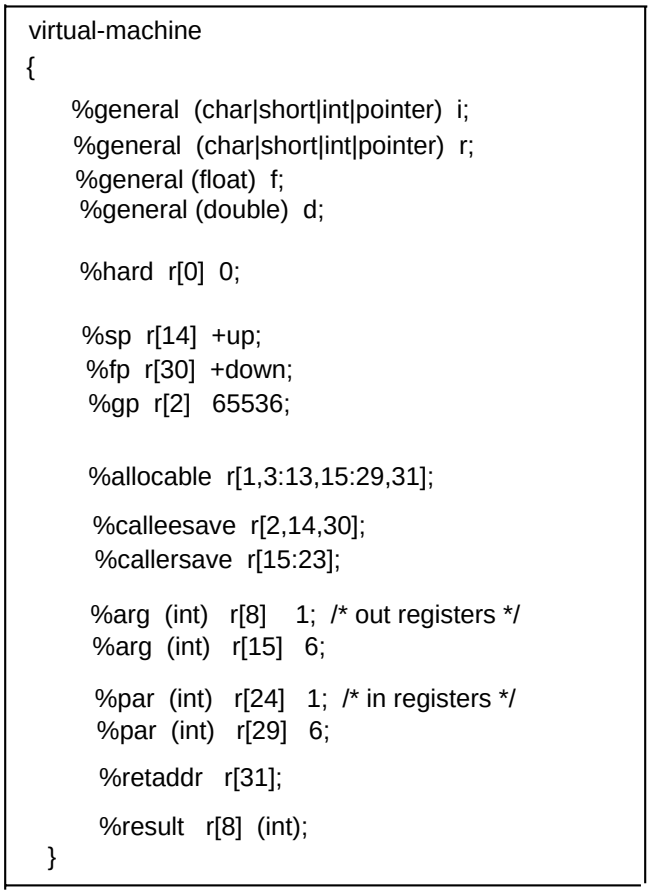
[5] *Pipeline de avanço explícito* é uma *pipeline* que retém seu estado até que instruções de um conjunto em particular sejam executadas.

[6] DAG de código é a estrutura de dados usada no processo de escalonamento de instruções. Representa-se nesta estrutura de dados os blocos básicos[7] em que foi dividido o programa.

de procedimento; definir o uso dos registradores. Por exemplo, na `SPARC` e na maioria das arquiteturas, durante a entrada de um procedimento, um registrador, o apontador de *frame*, é estabelecido para apontar para a base do registro de ativação corrente na pilha. Todas as referências a registradores locais utilizam índices a partir dele, de modo que os registradores locais mantêm os mesmos deslocamentos, mesmo se o apontador de pilha variar durante a execução do procedimento. Na máquina `MIPS` não existe um apontador de *frame*. Os registradores locais são endereçados relativamente ao `sp`. De posse destas informações e algumas outras apresentadas nesta seção é, portanto, possível descrever o funcionamento da máquina alvo. A Figura 3 mostra um trecho da especificação da máquina alvo para uma determinada arquitetura. As declarações e definições permitidas nesta seção são:

- Declaração de pilhas: %sp define o registrador usado como apontador de pilha.
- Definição do registro de ativação: %fp define o registrador usado como apontador do registro de ativação.
- Definição da área global: %gp representa o apontador global e a área usada.

- Declaração de registradores de propósito geral: %general indica os registradores de propósito geral para um dado tipo.
- Declaração dos registradores alocáveis: define através da diretiva %allocable os registradores que podem ser usados pelo alocador de registradores.
- Declaração de registradores com valores pré-definidos: %hard define registradores que contêm valores especificados pelo *hardware* da arquitetura em questão.
- Definição dos registradores preservados: define registradores com a diretiva %calleesave que devem ser preservados pelo procedimento chamado. Na arquitetura SPARC corresponde aos registradores (*out*) declarados pela diretiva %arg.
- Definição dos registradores preservados pelo procedimento chamador: define registradores com a diretiva %callersave que devem ser preservados pelo procedimento chamador. Na arquitetura SPARC corresponde aos registradores (*in*) declarados pela diretiva %par.
- Definição de argumentos específicos: define os argumentos do tipo especificado em %reg.
- Definição de parâmetros específicos: define os parâmetros do tipo especificado em %reg.
- Definição do resultado da operação: %result define o registrador que contém o resultado de acordo com o tipo especificado.



```
virtual-machine
{
    %general  (char|short|int|pointer)  i;
    %general  (char|short|int|pointer)  r;
    %general (float)  f;
    %general (double)  d;

    %hard  r[0]  0;

    %sp  r[14]  +up;
    %fp  r[30]  +down;
    %gp  r[2]   65536;

    %allocable  r[1,3:13,15:29,31];
    %calleesave  r[2,14,30];
    %callersave  r[15:23];

    %arg  (int)   r[8]    1;  /* out registers */
    %arg  (int)   r[15]   6;

    %par  (int)   r[24]   1;  /* in registers */
    %par  (int)   r[29]   6;

    %retaddr   r[31];

    %result   r[8]  (int);
}
```

Figura 3: Trecho da Especificação da Máquina Alvo para uma dada Arquitetura

- Definição do endereço de retorno: %retaddr define o registrador que contém o endereço de retorno.
- Definição do método de avaliação de argumento: a idéia da diretiva associada a este item é poder especificar quando expressões devem ser avaliadas ou não. A diretiva usada neste caso é %evalargs.

As informações coletadas das diretivas %calleesave, %callersave, %allocable, %args, %par, %result e %equiv são colocadas em uma matriz para serem usadas na geração de código, durante o escalonamento de instruções, na construção do `DAG` de interferência[8] de registradores e essencialmente na passagem de parâmetros. As demais diretivas são utilizadas durante a execução da máquina alvo.

[8]DAG de interferência é a estrutura de dados usada no processo de alocação de registradores.

Seção de Definição de Instruções
O objetivo desta seção é descrever as instruções da máquina alvo e suas funções. A seção de definição de instruções é composta de dois tipos de instruções: as instruções comuns e as instruções especiais. Cada diretiva %instr descreve uma instrução em cinco partes. A Figura 4 apresenta a sintaxe de uma instrução. A primeira parte especifica o mnemônico e os operandos da instrução. A segunda parte, entre parênteses, especifica, opcionalmente, as restrições de tipos dos operandos. A terceira parte, entre chaves, define a semântica da instrução. A quarta parte, entre colchetes, define os recursos utilizados pela instrução. A quinta parte, entre parênteses, define informações sobre o custo e os ciclos gastos pela instrução, etc.

```
%instr   fstoi  f, f  ($1 == float & $2 == int)
    {$2 = int($1); }
    [IF; ID; 5*IE; IW; ]
    (1,5,0)
```

Figura 4: Exemplo de uma Instrução em LDA

Incluem-se na categoria de instruções especiais as instruções para movimentação entre registradores, as declarações de instruções sem operação, as declarações das instruções "glue" e as definições de instruções para especificar uma nova latência. As instruções especiais auxiliam a compilação. Informações coletadas da diretiva %glue especificam transformações dependentes da máquina alvo. Elas são colocadas em uma tabela e posteriormente usadas durante o reconhecimento de padrão. A Figura 5 mostra um trecho da descrição da seção de declaração de instruções. A partir das informações coletadas nas três seções que compõem a LDA são derivadas várias tabelas. As mais importantes são (1) a tabela contendo os recursos utilizados pelas instruções cujo conteúdo da tabela de recursos é usado essencialmente na rotina básica de escalonamento de instruções, para verificar os conflitos existentes e agrupar instruções; (2) a tabela com informações sobre as instruções. Informações contidas nesta tabela são utilizadas durante a fase de inicializações do compilador e em várias funções do gerador de código. Por exemplo, nas rotinas de escalonamento de instruções, na rotina básica do escalonamento, na construção do DAG de código, na construção do DAG de interferência de registradores, na geração de código, durante a coloração do grafo de interferência, no reconhecimento de padrões, etc.

```
instructions
{
 %instr   cmp   r,  r,  r[0]    (;   clk-cc)
   {cc  =  ($1  ::  $2); }
    [IF; ID; IE; IW;]
    (1, 1, 0)


 %instr  be   #rel-lab
     {if  cc ==  0
           goto  $1;}
     [IF;  ID;  IE;]
     (1,  1,  1)

  %move   mov   i,  r
      {$2  =  $1;}
      [IF;  ID;  IE;  IW;]
      (1,  1,  0)

   %glue  r,   #uns-const13            (int)
        { ($!  ==  $2)  ==>  (($1  ::  $2) == 0);}
 }
```

Figura 5: Trecho da Especificação de Instruções em uma dada Arquitetura

3.1.4 Escopo de Aplicação de LDA

Completeza é uma qualidade desejável em uma linguagem para descrição de arquitetura para sistemas redirecionáveis. LDA ainda não cobre toda a classe das máquinas superescalares, contudo, ela é capaz de modelar a maior parte das características inerentes destas arquiteturas, abrangendo as características básicas das máquinas `RISC`. Para ter uma idéia de sua aplicabilidade apresentamos algumas das características mais comuns às arquiteturas superescalares, comercialmente disponíveis. As diretivas de LDA são suficientes para descrever os seguintes aspectos das arquiteturas superescalares:

Conjunto de registradores compartilhados:
Na especificação da arquitetura em questão, o projetista do compilador pode declarar vários conjuntos de registradores, bem como registradores com propósitos específicos. Para isto ele tem à sua disposição a diretiva %reg.

Pares de registradores:

O projetista do compilador pode declarar através da diretiva %reg um conjunto de registradores para representar o par e indicar que este conjunto se sobrepõe a outro conjunto de registradores utilizando a diretiva de especificação %equiv. As funções de escape, definidas pelo projetista, permitem que se tenha acesso a metade de registradores.

Janela de registradores:

A janela de registradores, visível por uma função em particular, é um subconjunto do conjunto total de registradores. O projetista do compilador usa as diretivas %arg e %par para especificar separadamente os argumentos e parâmetros, o que modela a renomeação de registradores, e, utiliza as diretivas %calleesave e %callersave para modelar o salvamento dos registradores especificados.

Registradores com valores pré-definidos por *hardware*:

Os registradores com valores pré-definidos por *hardware* são especificados pela diretiva %hard.

Dependência Estrutural:

O projetista do compilador especifica as dependências de recursos ou dependências estruturais explicitamente durante a especificação de cada instrução da arquitetura em questão. Durante o escalonamento de instruções verificam-se estes recursos. Isto evita atrasos por dependência estrutural. Os recursos devem ter sido préviamente definidos na seção de declarações através da diretiva %resource.

Esquema de prioridade por *hardware* para contenção de recursos:

O projetista do compilador especifica as prioridades dos recursos associando valores numéricos aos recursos pertinentes na seção de declarações e a instrução indica sua prioridade usando o recurso. A prioridade do recurso é verificada durante o escalonamento de instruções.

Carga de ponto flutuante de precisão dupla usando duas instruções:

A facilidade das funções de escape permite ao projetista do compilador escrever uma função em linguagem C para gerar uma seqüência de duas instruções, com cada uma delas tendo acesso a uma metade de um registrador de precisão dupla. As duas instruções de carga são então escalonadas como instruções separadas.

Desvios com *delay slots*:

Durante a especificação de cada instrução da máquina pertinente o projetista do compilador indica o número de instruções que devem ser colocadas no código para que a instrução de desvio seja executada sem causar atrasos na *pipeline*. Com esta informação *no-ops* são inseridas no código escalonado.

Desvios com *delay slots* executados condicionalmente:

O projetista do compilador pode especificar um *delay slot* com valor negativo na diretiva da instrução para indicar que a instrução no *slot* só é executada se o desvio ocorrer.

Latência de operação dependente do destino da instrução:

O projetista do compilador especifica um par de instruções com restrições e uma nova latência de operação por meio da diretiva %aux. Caso as restrições sejam satisfeitas,

substitui-se o valor do rótulo no vértice do DAG de código entre as duas instruções pelo valor da nova latência.

Pipelines **com avanço explícito:**
O projetista do compilador pode especificar esta característica da arquitetura `i860` utilizando-se das diretivas %reg, temporal, %clock, %element e %class.

Registradores de código de condição:
O registrador de código de condição pode ser declarado a parte como um registrador temporal utilizando as diretivas %reg e temporal. As diretivas das instruções indicam o resultado da comparação utilizando este registrador. Porém, diretivas separadas devem ser especificadas na descrição da máquina para as instruções que ligam o código de condição como um efeito colateral, por exemplo, subtração.

3.2 Ambiente da LDA

A LDA apresentada na Seção 3.1 faz parte de um sistema gerador de geradores de código para arquiteturas superescalares (GGCO). O GGCO recebe como entrada a especificação de uma arquitetura de máquina em LDA e gera uma série de tabelas e funções que representam o resultado do processamento das informações extraídas das principais diretivas de LDA. Estas tabelas e funções constituem a parte dependente da arquitetura em análise.

4 Conclusão

Este relatório apresentou várias abordagens para geração e otimização de código que se utilizam de descrição de arquiteturas de máquinas alvo para atingirem seus objetivos. Foram mostradas também, por meio de exemplos, algumas notações utilizadas nas especificações das máquinas, seguidas de uma análise das mesmas. Desta análise, conclui-se que, é difícil chegar a um consenso sobre qual linguagem é a mais adequada, porque quase todas diferem basicamente, apenas na notação e terminologia adotada, tornando a seleção da "melhor" apenas uma questão de gosto. Entretanto, certas características presentes ou ausentes em algumas linguagens, apontadas na seção 2.8, auxiliam na seleção. Resumidamente, uma linguagem para descrição de arquiteturas, tendo em vista a geração automática de otimizadores de código para arquiteturas CISC, deve, em princípio, possuir as seguintes propriedades:

1. possuir mecanismos para definição dos modos de endereçamento;
2. permitir definição completa da semântica das instruções, incluindo atualização de código de condição;

3. ser concisa e com alto poder de expressão.

Entre as abordagens apresentadas, três delas, [Davidson, 1981], [Kessler, 1984] e [Costa, 1990], destacam-se das demais, por sua simplicidade, pelo seu poder de expressão e pela forma concisa de descrever as arquiteturas de máquinas. Os sistemas *PO* e a primeira versão de *PEEP*, atribuídos a Davidson e Robert Kessler, respectivamente, foram desenvolvidos em paralelo, e baseiam-se na descrição de máquina para atingirem seus objetivos. Entretanto, as informações sobre a linguagem utilizada por Robert Kessler são mínimas, o que dificulta uma possível análise. Em conclusão, os trabalhos de Costa e Davidson satisfazem os requisitos citados acima, exceto por:

1. A descrição de Davidson não dispõe de mecanismos para definir custos.
2. A descrição do repertório de instruções de Costa é incompleta porque não possui mecanismos para indicar operação com os códigos de condição.

O conhecimento do custo é necessário para dirigir o processo de otimização, e o efeito completo das instruções deve estar claramente especificado, para que uma substituição por outra sequência de instrução proceda. Este é um requisito fundamental se o objetivo almejado é otimizar o código gerado. Contudo, as abordagens de Costa e Davidson podem ser adaptadas, suprindo estas deficiências.

Para especificar as características dos processadores RISC, linguagens de descrição de arquiteturas com as propriedades descritas neste relatório para arquiteturas CISCs não são suficientes. Uma linguagem para estas máquinas deve possuir mecanismos que lhe permitam tratar o escalonamento de instruções de forma satisfatória.

Os sistemas que se aplicam a estas arquiteturas são poucos (veja a Figura 1 da Seção 2), e apenas um deles, o sistema Marion de Bradlle, possui uma linguagem de descrição de arquiteturas, embora incompleta.

Alguns sistemas existentes incluíram informações sobre o escalonamento de instruções de alguma forma. O sistema Mahler de Wall e Powell [Wall and Powell, 1987] utiliza informações sobre escalonamento, mas não possui uma linguagem de descrição de arquitetura explícita. Mahler utiliza uma linguagem de montagem para uma máquina virtual sem *pipeline* e com um número infinito de registradores. Detalhes sobre escalonamento estão contidos dentro do tradutor de Mahler e, portanto, uma substituição na arquitetura de máquina implica em substituições manuais no tradutor.

O compilador GNU C [Stallman, 1989] é considerado um compilador redirecionável bem sucedido. GNU utiliza descrições interpretadas de máquina, relembrando o sistema PO de

Davidson e Fraser. Muito embora GNU C tenha sido transportado para várias arquiteturas RISCs, a descrição de máquina não contém informações sobre escalonamento. Tiemann escreveu um escalonador para GNU C [Tiemann, 1989] onde latências dependentes da arquitetura alvo e informações sobre recursos computacionais são encapsulados. Entretanto, ele não indica em seu trabalho a forma do encapsulamento.

Bradlee [Bradlee et al., 1991] projetou o protótipo de um sistema de geradores de código. Este sistema é o único que possui uma linguagem de descrição de máquina embutida, entretanto as primitivas de sua linguagem descrevem apenas os componentes básicos dos processadores RISC. Ela não modela várias das características típicas das arquiteturas superescalares, como por exemplo: o esquema de prioridade do barramento destino usado em algumas arquiteturas para conter o uso de recursos; o mecanismo de janela de registradores; os efeitos colaterais das instruções, como o acionamento do registrador de condição e, principalmente o despacho múltiplo de instruções.

Para concluir, de todos os sistemas mostrados, vimos que todas as linguagens utilizadas possuem primitivas para modelar apenas características comuns aos processadores de uma determinada classe de arquitetura. Por exemplo, a linguagem de descrição ISP (*Instruction Set Processor*) [Bell and Newell, 1971] tem sido utilizada nas descrições de arquiteturas CISC. Entretanto, ela não é apropriada para processadores RISC, porque contém mais detalhes do que realmente é necessário em algumas áreas, enquanto pouco detalhe em outras. Para o gerador de código de uma máquina RISC, uma descrição detalhada do formato da palavra de estado do programa não é a questão mais crucial, sendo, por exemplo, o conhecimento das propriedades de escalonamento de cada instrução mais importante. Árvores e gramáticas também têm sido utilizadas pelos compiladores redirecionáveis para CISCs para descrever as arquiteturas de máquinas. Vários sistemas produzem sofisticados métodos de seleção de instruções para CISCs. Até esta data, a única linguagem de descrição existente projetada por [Bradlee et al., 1991] para atender máquinas RISC é deficiente para esta classe de processadores. Como mostrado nesta seção, ela só modela características básicas dos processadores RISCs.

Referências

[Aho and Ganapathi, 1985] Aho, A. V. and Ganapathi, M. (1985). Efficient tree pattern matching: An aid to code generation. In *Conf. Rec.* 12^{th} *ACM Symp. on Princ. of Programming Languages*, pages 334–340.

[Aho et al., 1989] Aho, A. V., Mahadevan, G., and Tjiang, S. W. K. (1989). Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4).

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compiler Principals, Techniques and Tools*. Addison Wesley Publishing Company.

[Aigrain et al., 1984] Aigrain, P. et al. (1984). Experience with a graham–glanville code generator. In *Proceedings of the Sigplan'84 Symposium on Compiler Construction*, pages 13–24. ACM Sigplan Notices 19(6).

[Bell and Newell, 1971] Bell, C. G. and Newell, A. (1971). *Computer Structures: Readings and Examples*. McGraw-Hill New York.

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

[Benitez and Davidson, 1991] Benitez, M. E. and Davidson, J. W. (1991). Code generation for streaming: an access/execute mechanism. In *ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara California.

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

[Cattell, 1980] Cattell, R. G. G. (1980). Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2).

[Cattell et al., 1979] Cattell, R. G. G., Newcomer, J. M., and Leverett, B. W. (1979). Code generation in a machine-independent compiler. In *ACM Sigplan - Conference Compiler Construction*.

[Chase, 1987] Chase, D, R. (1987). An improvement to bottom-up tree patern maching. In *Conference Record* 14^{th} *ACM Symposium on Principles of Programming Languages*, pages 168–177.

[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.

[Davidson, 1985] Davidson, J. W. (1985). Simple machine description grammars. Computer Science Technical Report 85-22, School of Engineering and Applied Science, Charlottesville Virginia.

[Davidson, 1986] Davidson, J. W. (1986). A retargetable instruction reorganizer. *ACM Sigplan Notices*, 21(7). Proceedings of the ACM Sigplan'86, Symposium on Compiler Construction.

[Davidson and Fraser, 1980] Davidson, J. W. and Fraser, C. W. (1980). The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2).

[Davidson and Fraser, 1984] Davidson, J. W. and Fraser, C. W. (1984). Register allocation and exhaustive peephole optimization. *Software-Practice and Experience*, 14(9):8857–865.

[Davidson W. and Fraser W., 1984] Davidson W., J. and Fraser W., C. (1984). Automatic generation of peephole optimization. *ACM Sigplan Notices*, 19(6). Proceedings of the ACM Sigplan'84, Symposium on Compiler Construction.

[Fraser, 1989] Fraser, C. W. (1989). A language for writing code generators. In *Sigplan'89 Conference on Programming Language Design and Implementation*, Portland Oregon.

[Fraser and Henry, 1991] Fraser, C. W. and Henry, R. R. (1991). Hard-coding bottom-up code generation tables to save time and space. *Software-Practice and Experience*, 21(1):1–12.

[Fraser W. and Wendt, 1986] Fraser W., C. and Wendt, A. L. (1986). Integrating code generation and optimization. *ACM Sigplan Notices*, 21(7). Proceeding of the ACM Sigplan'86 Symposium on Compiler Construction, Palo Alto, California, June 1986.

[Fraser W. and Wendt, 1988] Fraser W., C. and Wendt, A. L. (1988). Automatic generation of fast optimizing code generators. *ACM Sigplan Notices*, 23(7):79–84. Proceedings of the ACM Sigplan'88 Symposium on Compiler Construction.

[Ganapathi and Fischer, 1985] Ganapathi, M. and Fischer, C. N. (1985). Affix grammar driven code generation. *ACM Transaction Programming Language and Systems*, 7(4):560–599.

[Ganapathi and Fischer, 1992] Ganapathi, M. and Fischer, C. N. (1992). Description-driven code generation using attribute grammars. In *Proceedings of the 9th POPL Conference*, pages 108–119.

[Giegerich, 1983] Giegerich, R. (1983). A formal framework for the derivation of machine specific optimizers. *ACM Translations on Programming Languages and Systems*, 5(3):478–498.

[Graham et al., 1982] Graham, S. L. et al. (1982). An experiment in table driven code generation. In *ACM Proceedings of the Sigplan'82 Symposium on Compiler Construction*, Boston Massachusetts. ACM Sigplan Notices 17(6).

[Graham and Glanville, 1978] Graham, S. L. and Glanville, R. S. (1978). A new method for compiler code generation. In *In Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, Tucson Arizona.

[Graham and Pelegri-Llopart, 1988] Graham, S. L. and Pelegri-Llopart, E. (1988). Optimal code generation for expression trees: An application of burs theory. In *Conference Record of the* 15^{th} *ACM Symposium on Principle of Programming Languages*, pages 294–308.

[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.

[Henry and Damron, 1989a] Henry, R. R. and Damron, P. C. (1989a). Algorithms for table-driven code generators using tree-pattern matching. Technical Report # 89-02-03, Computer Science Department, University of Washington, FR-35 Seattle, WA 89195 USA.

[Henry and Damron, 1989b] Henry, R. R. and Damron, P. C. (1989b). Performance of table-driven code, generators using tree-pattern matching. Technical Report # 89-02-02, Computer Science Department, University of Washington, FR-35 Seattle WA 89195 USA.

[Johnson, 1978] Johnson, S. C. (1978). A portable compiler: Theory and practice. In *Proceeding of the 5th ACM Symposium of Principles of Programming Languages*, Tucson Arizona.

[Kessler, 1986] Kessler, P. B. (1986). Discovering machine-specific code improvements. *ACM Sigplan Notices.*

[Kessler, 1984] Kessler, R. B. (1984). Peep - an architectural description driven peephole optimizer. In *Proceedings of the Sigplan'84 Symposium on Compiler Construction*, pages 13–24. ACM Sigplan Notices 19(6).

[Leverett et al., 1980] Leverett, B. W. et al. (1980). An overview of the production- quality compiler-compiler project. *IEEE Computer*, 13.

[Rich, 1983] Rich, E. (1983). *Artificial Intelligence.* McGraw-Hill Book Company, New York.

[Russell, 1985] Russell, S. (1985). The compleat guide to mrs. Technical Report KSL-85-12, Stanford Knowledge Systems Laboratory, Stanford University.

[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.

[Wall and Powell, 1987] Wall, D. W. and Powell, M. L. (1987). The mahler experience: Using an intermediate language as the machine description. In *ASPLOS-II Proceedings - Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California.

[Warfield and Bauer, 1988] Warfield, J. W. and Bauer, H. R. (1988). An expert system for a retargetable peephole optimizer. *ACM Sigplan Notices*, 23(10).

[Wulf et al., 1975] Wulf, W. A., Johnson, R., Weinstock, C., Hobbs, S., and Geschke, C. (1975). *The Design of an Optimizing Compiler*. American Elsevier.