Departamento de Ciência da Computação

D C C

Guilherme Arantes - planeta água

Estudo de Técnicas de Compactação Aplicáveis a Linguagens Funcionais Visando Arquiteturas Superescalares

Aluna: Patrícia Campos Costa
Orientadora: Mariza Andrade da Silva Bigonha

Monografia de Projeto Orientado em Computação
Bacharelado em Ciência da Computação
Novembro de 1996
DCC-ICEx-UFMG

U F M G

Universidade Federal de Minas Gerais

RESUMO

O desejo de obtenção de processadores e execuções cada vez mais eficientes tem motivado a criação de novas arquiteturas e métodos de otimização de código. Este relatório apresenta uma revisão da bibliografia, salientando os aspectos mais importantes das arquiteturas superescalares e das linguagens funcionais. Mostra as vantagens e desvantagens relativas aos métodos de compactação existentes para linguagens imperativas. O relatório apresenta também um método de compactação conhecido por algoritmos pessimistas que é baseado na filosofia de *Software Pipelining*. Esse método realiza uma compactação global do programa e pode ser aplicado para programas escritos em linguagens funcionais, pode ser estendido para a compactação de funções recursivas em geral e é aplicável em arquiteturas superescalares.

ABSTRACT

The desire to have fasters processors and executions has motivated the designers of new architectures and methods for code otimization. This paper presents the most important aspects of the superscalar architectures and functional languages and the compaction methods used with imperative languages programs. It also shows a compaction method known as pessimistic algorithms based on software pipelining. The method makes a global compaction of the program and can be applied to programs written in functional languages, can be extended to the compaction of general recursive functions and can be used with superscalar machines.

SUMÁRIO

SUMÁRIO **4**

LISTA DE FIGURAS **5**

1 INTRODUÇÃO **6**

2 OBJETIVOS DO PROJETO **7**

3 ARQUITETURAS SUPERESCALARES **7**

3.1 GRAFO DE DEPENDÊNCIA 10
3.1.1 Dependência de Dados *10*
3.1.2 Dependência de Controle *11*
3.1.3 Dependência de Recursos *12*

4 LINGUAGENS FUNCIONAIS **12**

5 TÉCNICAS DE COMPACTAÇÃO **15**

5.1 COMPACTAÇÃO LOCAL 15
5.2 COMPACTAÇÃO GLOBAL 15
5.2.1 Métodos de Compactação Global *16*
5.2.1.1 Trace Scheduling 16
5.2.1.2 Percolation Scheduling 19
5.2.2 Métodos para Compactação de Estruturas de Controle *20*
5.2.2.1 Loop Unrolling 20
8.2.2.2 Software Pipelining 21
8.3 ALGORITMOS PESSIMISTAS 23
5.4 ANÁLISE DAS TÉCNICAS DE COMPACTAÇÃO 26

6 CONCLUSÃO **27**

7 BIBLIOGRAFIA **29**

LISTA DE FIGURAS

FIGURA 1: SEQÜÊNCIA DE CÓDIGO E UNIDADES DE TEMPO GASTAS PARA EXECUÇÃO DE CADA INSTRUÇÃO 8
FIGURA 2: DESPACHO DE INSTRUÇÕES : (A) EXECUÇÃO SEQÜENCIAL, (B)DESPACHO DE UMA INSTRUÇÃO A CADA CICLO DE MÁQUINA, E (C) DESPACHO DE MÚLTIPLAS INSTRUÇÕES. 9
FIGURA 3: PARALELISMO DE INSTRUÇÕES DE TIPOS DIFERENTES NAS UNIDADES FUNCIONAIS 9
FIGURA 4: GRAFO DE DEPENDÊNCIA. 10
FIGURA 5: REPRESENTAÇÃO DE APLICAÇÕES DE FUNÇÃO EMMIRANDA EM GRAFOS. 12
FIGURA 6: (A) FLUXO DE CONTROLE DE UM PROGRAMA E(B) TRECHO ESCOLHIDO COMO O MAIS PROVÁVEL DE SER EXECUTADO 16
FIGURA 7: TRECHO DE UM DAG PARA O CÓDIGO MOSTRADO ACIMA. 17
FIGURA 8: DAG DA FIGURA 7 APÓS MODIFICAÇÃO. 17
FIGURA 9: (A) TRECHO COMPACTADO E (B) O TRECHO REUNIDO AO PROGRAMA 18
FIGURA 10: TRANSFORMAÇÕES DO PERCOLATION SCHEDULE . (A) DELETE. (B) MOVE_OP. (C) MOVE_CJ. (D) UNIFICATION. 20
FIGURA 11: LOOP UNROLLING : (A) *LOOP* ORIGINAL , (B) *LOOP* DESENROLADO DUAS VEZES. 21
FIGURA 12: LOOP UNROLLING E SOFTWARE PIPELINING 22
FIGURA 13: ELIMINAÇÃO DE RECURSIVIDADE DE CAUDA. 27

1 Introdução

Linguagens funcionais consistem em definições de funções e aplicações de funções. O elemento principal nas linguagens funcionais é a expressão, enquanto que nas linguagens imperativas é o comando. Não existe o conceito de variáveis do ponto de vista de que variáveis podem mudar de valor no decorrer da execução do programa. Uma característica importante deste tipo de linguagem é que uma expressão possui um valor bem definido portanto, a ordem na qual a expressão é avaliada não afeta o resultado final. Outro ponto importante das linguagens funcionais advém do fato de que dada a sua natureza, elas oferecem mais oportunidades para explorar o paralelismo e outras facilidades inerentes às arquiteturas superescalares que as linguagens imperativas.

Uma das principais características das arquiteturas superescalares é a separação dos componentes do processador em unidades funcionais e a habilidade de despachar e executar mais de uma instrução por ciclo de máquina. Estas propriedades auxiliam na construção de um compilador permitindo que o mesmo produza um bom código para estas máquinas. O paralelismo de grão fino para esta classe de arquitetura denomina-se compactação. A compactação consiste em reconhecer e escalonar grupos de operações que podem ser executadas em paralelo. A compactação pode ser local, envolvendo trechos de programas sem desvios ou pode ser global abrangendo as operações de desvios.

Várias técnicas de compactação foram desenvolvidas para ambientes baseados em linguagens imperativas. Para compactação global, as mais importantes são *Trace scheduling* [Fischer, 1981] e *Percolation scheduling* [Nicolau, 1985]. Para a compactação de estruturas de controle, usa-se uma técnica denominada *software pipelining* [Lam, 1988]. Outras abordagens utilizadas em *loop* são: (i) desdobramento de *loops* (*loop unrolling*[GS, 1992, BGS, 1994]), caracterizada por construir um *loop* com um número menor de iterações, e (ii) algoritmos pessimistas [Ebcioglu, 1987] que efetuam a execução simbólica do programa tentando executar operações, o mais cedo possível, por meio do uso de um desdobramento de *loop* controlado.

Esta monografia está organizada da seguinte forma: a Seção 2 apresenta os objetivos do projeto. A Seção 3 apresenta alguns tópicos importantes relacionados com a classe dos processadores superescalares, como por exemplo:
(1) a identificação dos tipos de dependências existentes entre as instruções;
(2) a necessidade do grafo de dependências na avaliação da ordem em que as instruções serão despachadas;
(3) algumas técnicas para tratamento de desvios condicionais e incondicionais que podem ser utilizadas para que o despacho das instruções do desvio não afete a semântica do programa;
(4) recursos, como: janela de instruções, renomeação automática de registradores, etc.
A Seção 4 apresenta uma introdução às linguagens funcionais. A Seção 5 mostra as técnicas de compactação global e local dando ênfase à primeira e, ao final da seção, apresenta uma análise das técnicas abordadas nesta monografia. A Seção 6 apresenta a conclusão deste trabalho seguida da bibliografia consultada.

2 Objetivos do Projeto

Este projeto teve como objetivo o estudo dos métodos de compactação local e global existentes para linguagens imperativas e sua possível aplicação em linguagens funcionais usando arquiteturas superescalares. Como resultado final desta pesquisa apresentamos os pontos mais importantes de cada método estudado, suas vantagens e desvantagens. Apresentamos também um método capaz de compactar funções recursivas gerais, que pode ser aplicado aos programas escritos em linguagens funcionais, que se encaixa muito bem nas arquiteturas superescalares e que mostra ser possível o uso de algumas das técnicas estudadas com estas linguagens.

Para alcançar nossos objetivos, fizemos:

1. Uma revisão da literatura abrangendo os tópicos: arquiteturas superescalares, linguagens funcionais, técnicas de geração de código e métodos de compactação para linguagens imperativas.
2. Um estudo da viabilidade de adotar a visão pessimista da compactação [Ebcioglu, 1987] nos programas escritos em linguagens funcionais.
3. Investigamos a possibilidade de uma generalização natural da abordagem de *software pipelining* [Lam, 1988] para as funções recursivas, considerando que as funções *tail-recursive* [Pou, 1994] podem ser traduzidas diretamente em *loops* sem a introdução de dependências adicionais.
4. Estudamos a viabilidade de permitir que o processo de compactação fosse capaz de compactar funções *non tail-recursive* [Pou, 1994] porque programas em linguagens funcionais também possuem este tipo de função. Na verdade, este fato constituiu a principal diferença entre a pesquisa que nos propusemos e os trabalhos anteriores que aplicavam compactações somente aos *loops* encontrados em programas baseados em linguagens imperativas.

3 Arquiteturas Superescalares

O desejo de obtenção de processadores mais velozes, fez com que diversas técnicas para exploração do paralelismo existente nos diversos níveis hierárquicos que formam um sistema de computador fossem desenvolvidas. Dois tipos de paralelismos podem ser explorados, o paralelismo de alto e baixo nível.

O paralelismo de alto nível é encontrado onde dois ou mais processadores executam um mesmo trecho de programa. Este tipo de paralelismo é basicamente limitado pelo desempenho dos processadores e pela sua rede de interconexão.

Já o paralelismo de baixo nível explora o paralelismo existente no interior de um único processador. As arquiteturas superescalares surgiram graças ao estudo das técnicas de paralelismo de baixo nível e ao avanço da tecnologia. Elas exploram o paralelismo a nível de instrução, e se caracterizam pela presença de muitas unidades funcionais que podem operar em paralelo possibilitando a execução de mais de uma instrução por ciclo de máquina. Normalmente elas fazem, também, uso do mecanismo de *pipeline*. Uma *pipeline* pode ser definida como uma janela onde várias instruções seqüenciais

executam simultaneamente, normalmente em fases distintas, e não podem depender uma da outra.

Uma configuração típica desta classe de arquiteturas possui unidades de execução de ponto fixo e ponto flutuante independentes, muito embora também seja possível outras unidades mais especializadas. Como exemplos de arquiteturas superescalares têm-se as arquiteturas MC88100/MC88200 [Mot, 90] que possui quatro unidades funcionais distintas: para operações com inteiros, ponto flutuante, para instruções e dados. Outro exemplo, o RS/6000 da IBM [MBig, 92] possui, além das duas unidades citadas anteriormente, uma unidade de desvio. Todas as arquiteturas executam a maioria das instruções em um único ciclo de máquina e fazem uso intensivo de *pipeline*.

Nas arquiteturas superescalares, dois tipos de paralelismos podem ser explorados:
(i) aqueles relacionados com instruções do mesmo tipo que são executados simultaneamente em unidades funcionais iguais ou pela utilização de *pipelining* em cada unidade funcional.
(ii) Aqueles relacionados com instruções de tipos diferentes que podem ser executados nas diferentes unidades funcionais.
Tanto (i) como (ii) serão mostrados nos exemplos a seguir.

Ao sobrepor as instruções nesta família de arquiteturas, a *pipeline* aumenta a vazão das instruções. Contudo, o tempo total de execução de cada instrução individual não diminui e pode, inclusive, aumentar. Isto acontece devido ao código extra relacionado ao controle da *pipeline*. A Figura 2, baseada em um exemplo de [FDS, 1992], mostra a vantagem do uso da *pipelining* de instruções na execução da seqüência de código apresentada na Figura 1.

	Unidades de Tempo:
1. R0 := R1*R2;	5
2. R3 := R4+R5;	3
3. R6 := R7*R8;	5
4. R9 := R2+R6;	3
5. R1 := R1 and R4;	2

Figura 1: Seqüência de código e unidades de tempo gastas para execução de cada instrução.

A Figura 2(a) ilustra a execução deste trecho de programa sem paralelismo, ou seja, uma instrução de cada vez. Neste caso o programa leva 25 unidades de tempo até que todas as instruções sejam executadas. Já fazendo o despacho de instruções a cada unidade de tempo ou de múltiplas instruções ao mesmo tempo, de acordo com a disponibilidade de unidades funcionais, o tempo total gasto é de 12 unidades de tempo, como mostram os itens (b) e (c) da Figura 2.

Número da Instrução	Unidades de Tempo																					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1																						
2																						
3																						
4																						
5																						

(a)

Número da Instrução	Unidades de Tempo											
	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												

(b)

Número da Instrução	Unidades de Tempo											
	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												

(c)

Figura 2: Despacho de instruções : (a) execução seqüencial, (b)despacho de uma instrução a cada ciclo de máquina, e (c) despacho de múltiplas instruções.

A vantagem do paralelismo de instruções de tipos diferentes (ii) pode ser vista na Figura 3. O trecho de programa da Figura 1, executado em uma máquina que possui 2 unidades de adição, 2 unidades de multiplicação e 2 unidades lógica, leva 8 unidades de tempo ao invés de 25, como na Figura 2(a).

Unidade Funcional	Número da Instrução	Unidades de Tempo							
		1	2	3	4	5	6	7	8
Multiplicação	1								
Multiplicação	3								
Adição	2								
Adição	4								
Lógica	5								
Lógica									

Figura 3: Paralelismo de instruções de tipos diferentes nas unidades funcionais.

Neste pequeno exemplo, com exceção da instrução 4, todas as outras instruções puderam ser despachadas simultaneamente. Apesar de haver uma unidade de adição disponível para a execução da instrução 4, ela teve que ser atrasada pois era dependente da instrução 3. (Ver Seção 3.1)

É importante notar que existem alguns fatores que podem afetar o desempenho das arquiteturas superescalares, por exemplo: o número e tipo de unidades funcionais presentes, o tamanho da janela de instruções, o esquema de interconexão dos componentes, o número de instruções despachadas simultaneamente e o mal funcionamento da *pipeline*.

Além destes fatores, para obter uma boa utilização das unidades funcionais disponíveis e a *pipeline* sempre cheia, a ordem das instruções pode ter que ser modificada. Dois pontos são fundamentais na movimentação das instruções: uma boa utilização da máquina e a preservação da equivalência semântica do programa. A técnica de *software* utilizada para rearranjar seqüências de código durante a compilação com o objetivo de reduzir possíveis atrasos de execução, denomina-se escalonamento de instruções ou compactação.

3.1 Grafo de Dependência

As dependências forçam o seqüenciamento de instruções, diminuem o desempenho de processadores com paralelismo de baixo nível e, podem gerar três classes de conflitos na *pipeline*: dependência de dados, dependência de controle e dependência de recursos. As dependências de recursos, ou dependências estruturais, resultam da demanda por um mesmo dispositivo do processador no mesmo ciclo de máquina. As dependências de controle resultam do fluxo de controle do programa; as dependências de dados resultam do fluxo de dados do programa. Esta dependência provoca um atraso na *pipeline* quando o dado ainda não está disponível para uso. As instruções de desvios e, portanto as dependências de dados restrigem a eficácia da *pipeline*.

Destes conceitos de dependências advém o conceito de grafo de dependência. Este grafo é utilizado pelas diversas técnicas de compactação para analisar as dependências existentes entre os dados. Nele, os vértices representam instruções e uma aresta de i para j, por exemplo, indica que j depende de i. A Figura 4 [FDS, 1992] apresenta um exemplo de grafo dependência para o trecho de código a sua esquerda.

```
1.    A := 12;
2.    C := A+2;
3.    B := 127;
4.    D := A/4;
5.    E := D*A;
6.    F := A and B;
```

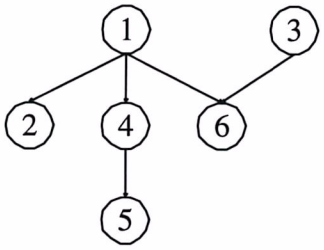

Figura 4: Grafo de dependência.

3.1.1 Dependência de Dados

Existem três tipos de dependências de dados.

1) Dependência de dados verdadeira: acontece quando uma instrução lê um valor antes que a instrução anterior tenha escrito o valor correto. Exemplo:

```
i: R1 := R3 and R0;
j: R2 := R1 *  2;          → j é verdadeiramente dependente de i
```

2) Anti-dependência: acontece quando uma instrução escreve em um registrador antes que a instrução anterior tenha lido o valor do registrador. Exemplo:

```
k: R1 := 127 and R0;
l: R0 := R2 * 2;           → l é anti-dependente de k
```

3) Dependência de dado de saída : acontece quando uma instrução escreve o resultado no registrador antes que a instrução anterior tenha escrito. Exemplo:

```
m: A := 12;
n:  B := A * 2;
o: A := 8;                    → o é dependente de dado de saída de m
```

Um recurso, que também está presente em algumas arquiteturas superescalares, é a técnica de renomeação automática de registradores. Esta técnica aloca, temporariamente, um outro registrador para uma instrução que apresenta falsa dependência ou dependência de saída em relação a uma instrução que ainda não foi concluída. Esta alocação é feita de tal forma que a instrução dependente pode ser despachada, sem ter que esperar a conclusão da instrução predecessora.

3.1.2 Dependência de Controle

As instruções de desvios diminuem o desempenho da *pipeline*. Elas fazem com que a mesma pare. Este fato ocorre quando o endereço do objeto do desvio não é conhecido no momento em que a unidade central de processamento está pronta para buscar a instrução de destino. Portanto, técnicas de tratamento de desvios condicionais devem ser especificadas. Alguns processadores identificam este problema por meio de um mecanismo de *interlock* por *hardware*. Outros processadores fazem uso de *software* para detectar e remover *interlocks*.

(1) *Interlock* por *hardware*: o mecanismo de *interlock* por *hardware* acarreta em um problema: introduz a necessidade de verificadores. Com isto as arquiteturas ficam mais complexas, e impõem uma sobrecarga no tempo de execução de todas as instruções. A eliminação deste *hardware* simplifica a arquitetura.

(2) *Interlock* por *software*: algumas das técnicas mais comuns para detectar e remover *interlocks* por *softwares* são:

(a) Muitas arquiteturas, entre elas as superescalares, evitam a dependências de controle usando um mecanismo chamado desvio com retardo. Um desvio com retardo é uma instrução de desvio que possui um ou mais *delay slots* seguindo o desvio [Mbig, 94]. Nesta estratégia geralmente usa-se a inserção de NOP's no programa, para neutralizar o efeito da busca antecipada e execução do próximo comando.

(b) Outra técnica movimenta a instrução de desvio n posições antes, de tal forma que ao ter que tomar a decisão de desviar, ou executar o próximo comando, já se sabe que atitude tomar. Porém, esta técnica só pode ser aplicada se as instruções precedentes não afetarem a avaliação da condição de desvio.

(c) Uma outra alternativa consiste em introduzir, logo após o desvio, uma cópia do trecho de programa formado pelos n comandos que iniciam com a instrução destino do desvio, *'then ou else'*, e rearranjar o endereço alvo do desvio. Contudo, esta técnica só pode ser aplicada se o endereço alvo for conhecido.

3.1.3 Dependência de Recursos

Este tipo de dependência aparece quando duas instruções necessitam de um determinado recurso de máquina durante o mesmo ciclo. Algumas arquiteturas possuem um mecanismo que viabiliza a análise antecipada das instruções a serem despachadas para alocação dos recursos de *hardware* disponíveis e o despacho de mais de uma instrução por ciclo de máquina. Este recurso, denominado janela de instruções, caracteriza-se por um conjunto de registradores que atua como um *buffer* e, é freqüentemente utilizado no escalonamento de instruções. Antes do despacho das instruções um pequeno trecho do programa em execução é colocado no *buffer* para que as instruções sejam analisadas antecipadamente.

4 Linguagens Funcionais

Programar em linguagens funcionais consiste em construir definições e expressões que serão avaliadas pelo computador. As expressões são as unidades principais da linguagem, elas são reduzidas, ou seja, transformadas de acordo com as definições, até que se chegue em uma forma imprimível. As linguagens funcionais são muito simples, concisas, flexíveis e poderosas.

Segundo [Ara, 1986], as características mais importantes das linguagens funcionais são:

- a ausência de variáveis ou efeitos colaterais.
- o programa é a uma função do ponto de vista matemático;
- a operação básica é aplicação de função: o programa é aplicado à entrada e o valor resultante é a saída do programa;
- o valor de uma expressão depende apenas do seu contexto textual, não da computação histórica, portanto, não existe conceitos de estado, contador de programa ou armazenamento presentes;

Um programa funcional tem como representação natural uma árvore ou grafo. A avaliação se processa por meio de passos simples que podem ser feitos aplicando-se reduções no grafo. A Figura 5 ilustra este processo. Dada a função: f x = (x +2) (x+4) em Miranda [Turner, 1985], que é uma linguagem funcional, a mesma pode ser representada como em 5(a). A aplicação de função f 4 em 5(b), por exemplo, resulta na árvore (c) . Executando a adição ou a subtração, em qualquer ordem, temos (d), e executando a multiplicação resulta em 0.

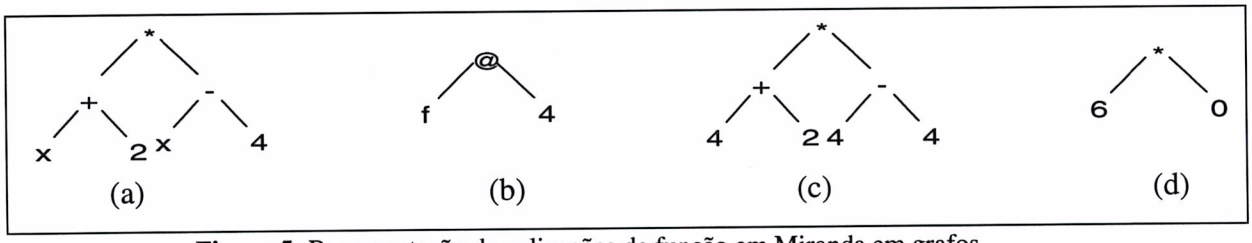

Figura 5: Representação de aplicações de função em Miranda em grafos.

As linguagens funcionais existentes são bem parecidas diferindo-se mais na sintaxe do que na semântica. Exemplos de linguagens funcionais são: SASL [Turner, 1976], ML

[Gordon, 1979], KRC [Turner, 1982], Hope [Burstall, 1980], LML [Augustsson, 1984], Miranda [Turner, 1985] e Orwell [Wadler, 1985].

Linguagens imperativas, como Fortran e Pascal foram implementadas para serem usadas em desenvolvimentos de programas nas arquiteturas de *Von Neumann*. Estas máquinas foram projetadas para executar operações seqüenciais em itens de dados escalares. Como conseqüência estas linguagens reforçam uma seqüência artificial na especificação de algoritmos. Essa seqüência não só adiciona verbosidade aos algoritmos como pode dificultar a execução do algoritmo em processadores superescalares ou arquiteturas paralelas.

As linguagens funcionais foram desenvolvidas com o propósito de permitir que um algoritmo seja capturado no programa, eliminando qualquer necessidade de inserção de detalhes não relacionados com o problema, evidenciando a correção do programa com uma análise mínima, sendo ao mesmo tempo independentes da máquina e de alto nível. Portanto, os programas escritos em linguagens funcionais são de mais alto nível do que aqueles escritos em linguagens imperativas.

Outras vantagens das linguagens funcionais em relação às linguagens imperativas são citadas em [Ara, 1986], por exemplo:

- são concisas e possuem semântica simples.
- São expansíveis e modulares.
- Sua notação compacta permite que mais algoritmo seja expresso por linha de código, o que aumenta a produtividade do programador. A experiência mostra que em um número maior de linhas a tendência é um número maior de erros.
- São livres de efeitos colaterais.
- Possuem facilidades de verificação, pois as provas podem se basear no conceito de função ao invés de serem baseadas no conceito de transição de estados.
- Oferecem mais oportunidades para explorar o paralelismo e outras facilidades inerentes as arquiteturas superescalares que as linguagens imperativas.
- A dependência de dados pode ser obtida da sintaxe, pois como as variáveis são modificadas apenas uma vez, não há dependência entre dois armazenamentos.
- Uma expressão possui um valor bem definido, o que faz com que a ordem da avaliação não afete o resultado final, podendo ser feita em paralelo.

Apesar das vantagens citadas, as linguagens funcionais possuem um custo de ineficiência em termos de execução pois vários fatores contribuem para degradar a execução de um programa funcional. Por exemplo:

- a ausência de uma arquitetura funcional. As arquiteturas de V*on Neumann* não são adequadas para a execução de programas em linguagens funcionais. Mas como não existe uma máquina funcional, uma forma de compactar linguagens funcionais é traduzí-la para linguagens imperativas e então aplicar as transformações no grafo de controle de fluxo obtido.
- Os programas fazem grande manipulação de listas necessitando *garbage collection* [Ara, 1986].
- A falta de atualização destrutiva, pois uma nova cópia da variável sempre é gerada quando a estrutura é modificada [Ara, 1986].
- Em geral os sistemas são mais interpretados do que compilados.

As linguagens funcionais podem, ainda, ser caracterizadas como linguagens de redução visto que a execução de programas funcionais consiste em reduzir as aplicações de funções aos seus resultados. Duas abordagens podem ser adotadas: redução de *strings* ou redução de grafos. Na redução de *strings* cada aplicação de função é avaliada e seu resultado é substituído no *string* que é a representação da expressão. As expressões não são compartilhadas e em cada ocorrência da expressão, a mesma redução deve ser feita. Na redução de grafos, os resultados das aplicações das funções são substituídos no grafo e é possível ter referências para as expressões já calculadas anteriormente evitando uma nova avaliação, contudo aqueles nodos que deram origem às transformações efetuadas e que não são mais necessários, precisam ser retirados por meio de um mecanismo de *garbage collection*. Uma análise das duas abordagens em relação ao tempo de execução e ao espaço gasto nos diria que, na redução de *strings* temos a utilização de menos espaço e mais execução e na redução de grafos utilizamos mais espaço e menos execução.

Outro fator importante na determinação da eficiência da execução de programas funcionais é a escolha da próxima aplicação a ser reduzida. Há duas possibilidades: mais interna ou mais externa. Na redução da aplicação mais externa, uma instrução é executada somente quando seu resultado é necessário para uma outra redução, sendo assim mais compatível com a abordagem redução de grafos. Por outro lado, uma redução mais interna, é mais compatível com a técnica de redução por *strings* já que uma instrução só é executada quando todos os seus argumentos já tiverem sido avaliados.

Levando em consideração os fatos expostos nos parágrafos anteriores, a escolha da técnica que propicia uma execução mais eficiente pode variar de acordo com as características da arquitetura utilizada e da classe de programas a serem executados. Uma das principais características das arquiteturas superescalares é separação dos componentes do processador em unidades funcionais e a habilidade de despachar e executar mais de uma instrução por ciclo de máquina. Se as diferentes unidades funcionais de uma máquina nesta família de arquiteturas tiverem mais de uma unidade com a mesma função, a redução por *strings* pode ser mais eficiente, pois várias aplicações de funções poderiam ser executadas ao mesmo tempo em unidades funcionais idênticas e substituídas nos *strings* sanando assim, o problema da execução excessiva que este tipo de redução apresenta. A redução de grafos também poderia tirar proveito das unidades funcionais idênticas executando em paralelo as operações replicadas evitando a inserção de novos ponteiros no grafo. Contudo seria indispensável, a presença de um mecanismo que facilitasse a operação de *garbage collection* na arquitetura para tornar a técnica de redução de grafos mais eficiente. Levando em conta que as arquiteturas superescalares, em geral, não possuem uma replicação de unidades funcionais idênticas e que estas arquiteturas não possuem mecanismos que facilitem *garbage collection*, então, não podemos afirmar qual dos dois métodos é mais eficiente para esta classe de arquiteturas, a escolha de um deles vai depender das características de cada arquitetura isoladamente. Podemos, apenas, concluir que um fator importante para o aumento da eficiência de execução da técnica de redução de *strings* em uma arquitetura superescalar depende do número de unidades funcionais idênticas enquanto que a redução de grafos exige, também, a presença de um mecanismo *garbage collector*.

5 Técnicas de Compactação

As técnicas de compactação usadas para explorar o paralelismo de baixo nível nas arquiteturas superescalares podem ser desenvolvidas em *hardware* ou *software* e se classificam em global e local. Para que se faça um bom uso dos recursos disponíveis na arquitetura é necessário a:

- construção do grafo de dependências,
- descrição dos recursos da máquina e
- empacotamento das instruções paralelas.

5.1 Compactação Local

A compactação local se limita a blocos básicos. Um bloco básico é um trecho de programa que não possui pontos de entrada, só se for o primeiro e nem pontos de saída, só se for o último, ou seja, não existem instruções de desvios condicional ou incondicional em seu interior. Este tipo de compactação não explora o paralelismo existente entre os blocos básicos. Após a divisão do programa em blocos básicos, os algoritmos propostos para este tipo de compactação [FER, 1984], [HG, 1982], [Pin, 1988] tentam empacotar, agrupar, as instruções a serem despachadas, verificando a possibilidade da execução de comandos de um mesmo bloco básico paralelamente. Como tudo é feito dentro do contexto do bloco básico, usando somente a compactação local é possível perder oportunidades de mover instruções de um bloco para outro, o que traria um ganho significante. Contudo, a compactação local pode ser usada juntamente com a compactação global para se obter melhores resultados. Outras técnicas desenvolvidas visando a compactação local, podem ser encontradas em [Bradlee et al., 1991], [Gibbons et al., 1986] e [Goodman and Wei-Chung-Hsu, 1988]. Neste texto nos focalizaremos na compactação global.

5.2 Compactação Global

A compactação global explora o paralelismo do programa todo, abrange os desvios e pode aumentar a taxa de concorrência transferindo instruções de um bloco básico para outro. Este método consiste na construção do grafo de fluxo de controle, sua análise e movimentação das instruções obedecendo os critérios de dependências estabelecidas no grafo.

Nas próximas seções serão apresentados alguns métodos desta abordagem. Eles estão divididos em duas categorias: os métodos de compactação global: *Trace Scheduling* [Fisher, 1981], *Percolation Scheduling* [Nic, 1985], e os métodos para compactação de estruturas de controle: *Software Pipelining* [Lam, 1988], e, *Loop Unrolling* [GS, 1992, BGS, 1994]. Apresentamos também um método, denominado algoritmos pessimistas que se encaixa nas duas categorias e que mostra a aplicabilidade de algoritmos desta natureza em programas escritos em linguagens funcionais.

5.2.1 Métodos de Compactação Global

5.2.1.1 Trace Scheduling

O *Trace Scheduling* [Fisher, 1981] é uma técnica de compactação global, que explora o paralelismo de trechos de programas denominados *traces*. Ele funciona da seguinte forma: inicialmente constrói-se o grafo, DAG (*Directed Acyclic Graph*) para a análise de dependência das instruções do programa a ser compactado. A partir do DAG construido o *Trace Scheduling*, basicamente, compõe-se de 4 fases: escolha do trecho, pré-processamento do trecho, escalonamento das instruções do *trace* e pós-processamento do trecho.

a) Escolha dos trechos:

Um fator determinante na qualidade do código resultante é justamente a escolha do trecho. O trecho deve ser aquele com mais probabilidade de ser executado e é escolhido usando métodos heurísticos. Uma possível heurística consiste em atribuir um valor para cada instrução de acordo com a probabilidade dela ser executada. Os valores podem ser obtidos através da execução prévia do programa não compactado para um conjunto típico de dados, ou podem ser passados pelo programador que conhece os trechos mais possíveis de serem executados. Se o algoritmo não consegue detectar corretamente o trecho mais provável de ser executado, a máquina executará operações desnecessárias e o desempenho será degradado. Esta fase pode ser vista na Figura 6 extraída de [FDS, 1992]. A Figura 6(a) mostra o grafo de fluxo de controle de um programa que será compactado. Os vértices representam instruções. A Figura 6(b) mostra o trecho de programa que foi selecionado pelo algoritmo.

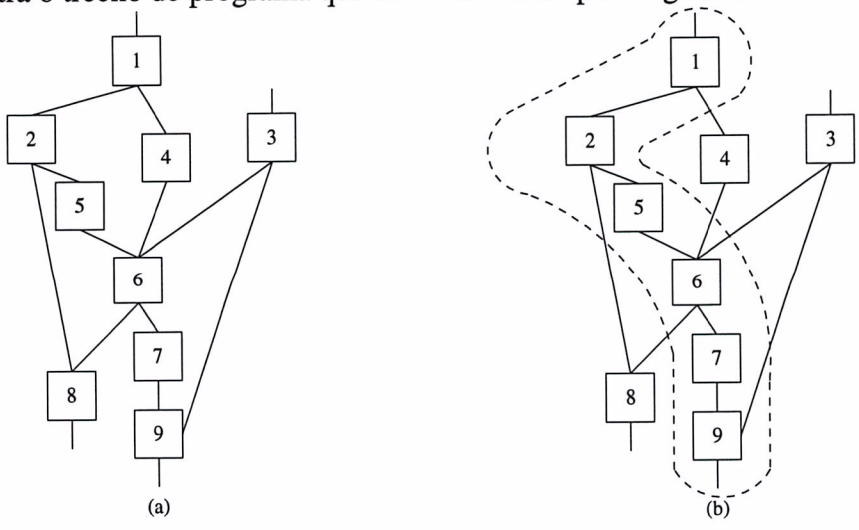

Figura 6: (a) Fluxo de controle de um programa e (b) Trecho escolhido como o mais provável de ser executado.

b)Pré-processamento do trecho:

Nesta fase, novas arestas são incluidas no DAG. Estas arestas conectam blocos de instruções com desvio condicional a blocos com instruções que neutralizam o efeito da execução antecipada e indevida de comandos precedentes a outros desvios. Por exemplo, dado o trecho de código a seguir, extraído de [FV, 1991], o DAG correspondente é apresentado na Figura 7:

```
...
a := b*c;
if x>0 then
  d := a-3
else
  f := a*3
  g := d+2
end;
...
```

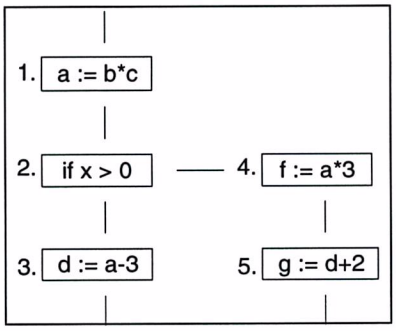

Figura 7: Trecho de um DAG para o código mostrado acima.

Assuma que o *trace* escolhido seja 1, 2 e 3, e que o gerador de código decida mover a operação 1 para depois do desvio 2. Se isto for feito, a instrução 4 que lê o valor de a, modificado na operação 1, irá ler um valor errado de a. Assim, o DAG da Figura 7 deve ser modificado como mostrado na Figura 8.

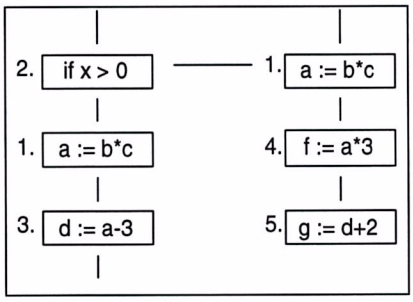

Figura 8: DAG da Figura 7 após modificação.

c)Escalonamento das instruções do trace:

Nesta fase, é feito o escalonamento das instruções propriamente dito, usando o algoritmo *List Scheduling* [Fis, 1981], que funciona da seguinte maneira: o

escalonamento mantém uma lista de instruções que estão aptas para serem escalonadas sem provocar atrasos. A cada iteração, seleciona-se uma instrução que já está pronta para ser escalonada de acordo com a heurística adotada e atualiza-se a lista. Uma instrução só é colocada em determinado ciclo de tempo se é compatível com os recursos disponíveis naquele ciclo.

As heurísticas adotadas para atribuir prioridade às instruções a serem escalonadas podem ser bem diversas, mas usualmente, a prioridade é determinada pelo maior caminho a partir do vértice em análise até a folha do DAG. O raciocínio por trás desta heurística é que quanto maior o comprimento, mais prioridade o vértice deve ter. Uma outra heurística usa o número de filhos de um nodo, quanto mais filhos, maior oportunidade de escalonar mais instruções, e por isso ele deve ter uma prioridade maior.

d)Pós-processamento do trecho:

Esta fase é responsável pela correção dos efeitos provocados pela movimentação de alguns comandos e pela inserção de blocos básicos para conectar o *trace* ao restante do programa. Estes novos blocos básicos atuarão como interface entre os pontos de entrada e os pontos de saída do *trace* compactado. Esta fase pode ser observada na Figura 9 onde temos em (a) o trace compactado "separado" do resto do programa e em 9(b) o trace compactado com os blocos de entrada e de saída inseridos ligando o trecho ao resto do programa. Os pontos de entrada e de saída estão representados na figura por meio dos blocos básicos R e S.

Como nesta técnica, cada trecho compactado é marcado para não ser compactado novamente, a exploração do paralelismo de *loops* pode ser feita antes da compactação dos *traces*. Assim, os *loops* não serão escolhidos como *trace*, pois já estarão marcados, não trazendo nenhum problema para a compactação.

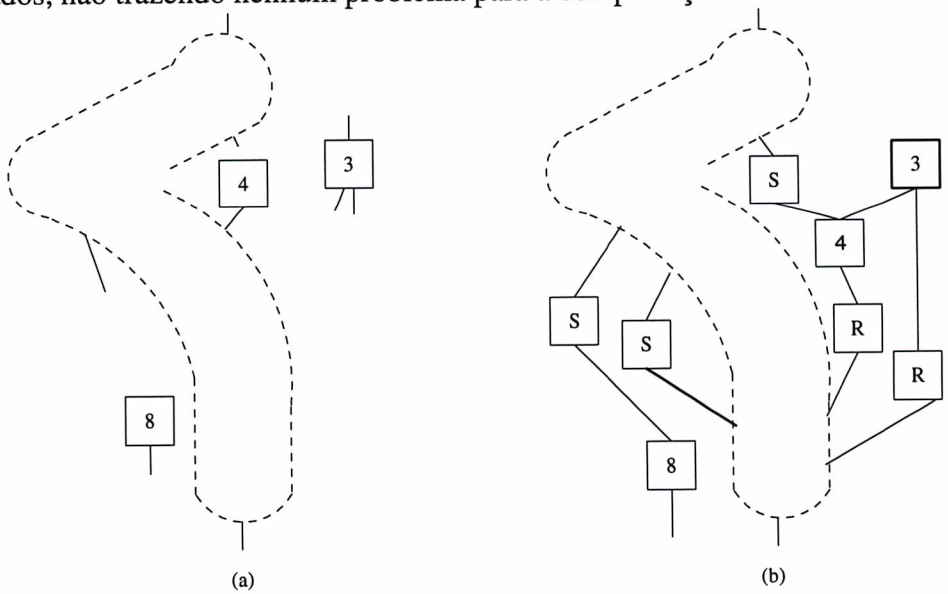

Figura 9: (a) Trecho compactado e (b) o trecho reunido ao programa

5.2.1.2 Percolation Scheduling

A técnica *Percolation Scheduling* [Nicolau, 1985] foi desenvolvida na tentativa de acabar com algumas deficiências da técnica *Trace Scheduling* [Fisher, 1981]. Entre elas:

(i) a compactação de somente trechos isolados do programa. Este fato faz com que o paralelismo não seja totalmente explorado,
(ii) o custo das inserções de blocos de reparo,
(iii) a compactação de somente um *trace* de cada vez,
(iv) a reduzida interação do usuário com o processo de compactação;
(v) a falta de evidência da eficiência para aplicações científicas.

Para superar estas deficiências, foi desenvolvido um ambiente interativo para a geração do código paralelo, reorganizando-se globalmente os programas de aplicação, visto que a movimentação de comandos é feita além das fronteiras dos blocos básicos.

O objetivo do algoritmo de *Percolation Scheduling* [Nicolau, 1985] é maximizar o paralelismo movendo operações de nodo para nodo de modo a maximizar o número de operações paralelas no DAG final. O ambiente provê primitivas de transformações que fazem alterações no DAG de acordo com as dependências presentes, bem como primitivas que realizam operações inversas, preservando a equivalência semântica do programa. Estas primitivas são:

1)Delete: elimina um vértice do grafo. Arestas dos antecessores do vértice removido são dirigidas para seu sucessor para neutralizar o efeito da operação. A Figura 10 (a) ilustra o efeito desta primitiva na eliminação de um vértice vazio do grafo.

2)Move_op: movimenta um comando para o vértice adjacente anterior. A Figura 10 (b), ilustra o efeito desta primitiva movimentando um comando *b* de um vértice *n* para seu antecessor *m*.

3)Move_cj: movimenta um comando de desvio condicional para um antecessor. A Figura 10(c) ilustra o efeito da operação move_cj, movimentando o comando de desvio condicional *x* de um vértice *n* para um vértice *m*.

4)Unification: unifica diversos comando idênticos. A Figura 10 (d) ilustra esta transformação, movimentando o comando *x* repetido em vários vértices para o vértice anterior.

Utilizando-se estas primitivas, a ordem dos comandos do programa a ser compactado pode ser modificada preservando-se a equivalência semântica em todos os caminhos do grafo que são afetados pelas transformações. Aplicações sucessivas das primitivas migram os comandos em direção ao topo do grafo que representa o programa, permitindo o empacotamento dos comandos do programa. Estas primitivas têm sido usadas como valiosas ferramentas em diversos experimentos. Este método se encaixa muito bem na execução de linguagens funcionais através de redução de grafos, pois uma vez construido o grafo a obtenção do DAG é bem natural.

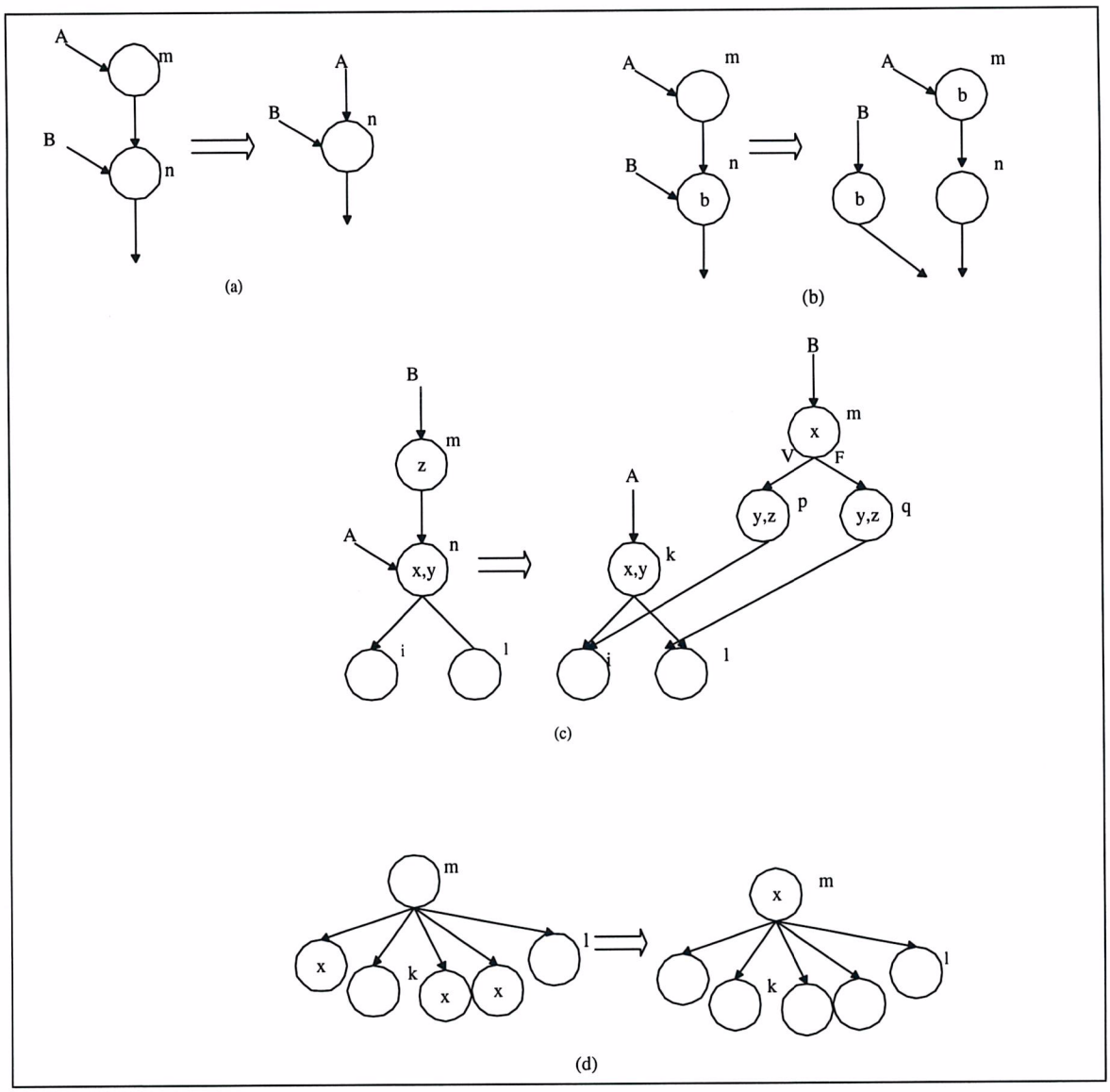

Figura 10: Transformações do Percolation Schedule . (a) Delete. (b) Move_op. (c) Move_cj. (d) Unification.

5.2.2 Métodos para Compactação de Estruturas de Controle

5.2.2.1 Loop Unrolling

A técnica de *Loop Unrolling* [GS, 1992, BGS, 1994] consiste em construir um *loop* com um menor número de iterações, repetindo-se o corpo do *loop* original seqüencialmente. Com este esquema temos um maior número de operações dentro do *loop*, eliminando ciclos de código extra dentro do *loop* original. Como conseqüência, as instruções do *loop* podem ser escalonadas de modo a maximizar o uso das unidades funcionais diponíveis, aumentando o paralelismo de instruções. As desvantagens desta técnica são: o tamanho do código gerado é maior e são necessários mais registradores para armazenarem resultados intermediários. A Figura 11, extraída de [BSG, 1994], mostra a melhora do desempenho com o desdobramento do *loop* duas vezes. Em 10(b) a sobrecarga do *loop* é reduzida à metade, pois duas iterações são executadas antes do

teste e o paralelismo de instruções é aumentado, pois a segunda atribuição pode ser executada enquanto os resultados da primeira estão sendo armazenados e as variáveis do *loop* estão sendo atualizadas.

```
do i=2, n-1
   a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a)

```
do i=2, n-2, 2
   a[i] = a[i] + a[i-1] * a[i+1]
   a[i+1] = a[i+1] + a[i] * a[i+2]
end do
      if (mod(n-2,2) = 1) then
  a [n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b)

Figura 11: Loop Unrolling : (a) *loop* original , (b) *loop* desenrolado duas vezes.

8.2.2.2 Software Pipelining

O algoritmo proposto por [Lam, 1988] utiliza uma técnica de redução hierárquica combinada ao *Software Pipelining* possibilitando a aplicação do *Software Pipelining* aos *loops* mais internos inclusive aqueles com desvios condicionais. A abordagem proposta escalona o programa hierarquicamente começando com as estruturas de controle mais internas. Assim que cada construção é escalonada, a construção é toda substituída por um nodo. O processo estará completo quando o programa estiver reduzido a um simples nodo. Esta técnica se aplica muito bem ao método de execução de linguagens funcionais se considerarmos a redução de grafos, pois na redução de grafos, também as funções vão sendo executadas e substituidas no grafo até que se chegue a um simples nodo de forma análoga ao *Software Pipelining*.

O objetivo da técnica *Software Pipelining* [Lam, 1988] é tentar reduzir o atraso provocado por acessos à memória para a obtenção dos operandos necessários à execução de uma operação em uma unidade funcional. Ela faz a transferência das instruções de acesso à memória para iterações anteriores ao *loop* de modo que os operandos estejam diponíveis em registradores no momento em que a operação for executada. As operações de uma iteração do *loop* são quebradas em *s* estágios e uma única iteração executa o estágio 1 da iteração *i*, o estágio 2 da iteração *i*-1, e assim por diante, analogamente ao *hardware pipelining*, que foi apresentado na Seção 3. A técnica reduz o intervalo entre iterações consecutivas do *loop*, diminuindo seu tempo total de execução. Ela reduz o custo do *startup* de cada iteração do *loop*, apesar de necessitar de código de *startup* antes do *loop* inicializar as primeiras iterações e código de *cleanup* depois do *loop* para drenar o *pipeline* para as últimas iterações.

As técnicas de *software pipelining* apresentam a possibilidade de produzir um código melhor com tempo de compilação menor que outras técnicas de escalonamento. Por exemplo a Figura 12, extraída de [ANN, 1996], ilustra este fato, comparando-a com *Loop Unrolling* [GS, 1992, BGS, 1994].

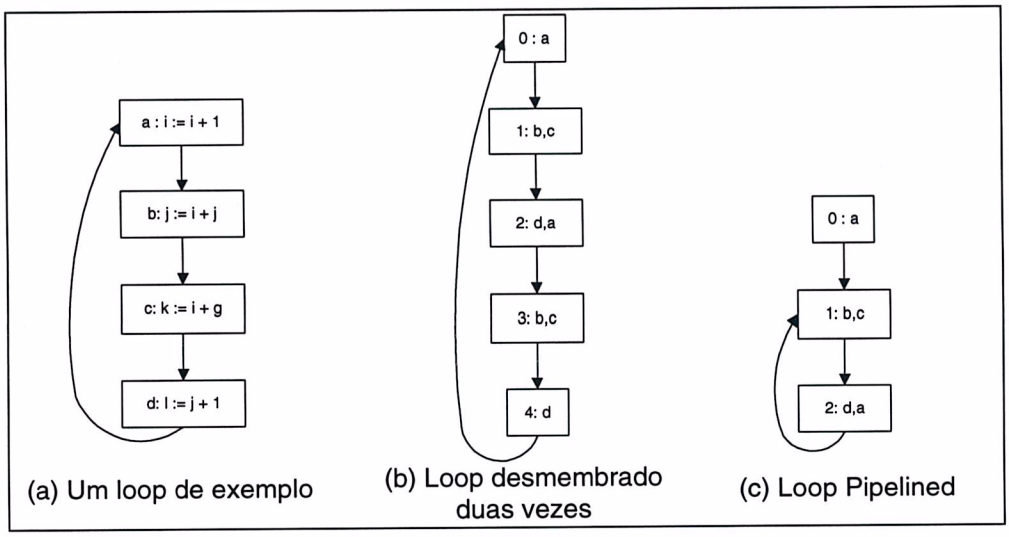

Figura 12: Loop Unrolling e Software Pipelining

Em (a) temos o *loop* original, em (b), é mostrado o resultado da aplicação da técnica *loop unrolling*, Seção 5.2.2.1, onde o *loop* foi desmembrado duas vezes. Em (c) é apresentado o resultado da aplicação de *software pipelining*. Observando o item (c) percebe-se que este *loop* possui dois tipos de paralelismos: um paralelismo dentro do corpo do *loop*, onde b e c podem ser executados simultaneamente, e um paralelismo entre as iterações do *loop*, onde o d de uma iteração pode sobrepor a da próxima iteração. A abordagem de *loop unrolling* usada em (b) permite que o paralelismo seja explorado entre algumas iterações do *loop* original, mas existe uma seqüência imposta entre as iterações do corpo do *loop* desmembrado. Se o *loop* tivesse sido totalmente desmembrado, todo o paralelismo interno ao *loop* e entre as iterações do *loop* poderia ter sido explorado, entretanto o desmembramento total é em geral impossível ou impraticável em determinados casos onde o tamanho do código cresceria muito. Já em (c) pode ser observado que o algoritmo de *software pipelining* provê uma maneira direta de explorar o paralelismo presente dentro do *loop* e entre suas iterações de tal forma que o mesmo atinge o efeito de um completo desmembramento.

Além da abordagem de *Software Pipelining* apresentada, existem na literatura várias outras implementações possíveis, por exemplo:

[ANN, 1996] apresenta um algoritmo para o *Software Pipelining* que contabiliza as restrições dos recursos de máquina de uma maneira suave integrando a gerência de restrição de recursos com o *software pipelining*. O algoritmo é formado por dois componentes: um escalonador e um analisador de dependências. O escalonador é usado para construir um *loop* paralelizado a partir de um *loop* seqüencial. Para cada instrução paralela, o escalonador seleciona operações a serem escalonadas baseado no conjunto de operações disponíveis para escalonamento naquela instrução e recursos disponíveis. O conjunto de operações disponíveis é mantido pelo analisador de dependências globais. Após a construção do *loop* paralelizado, o algoritmo de *software pipelining* verifica por estados que podem ser *pipelined*.

[WE, 1993] propõe uma técnica de *software pipelining* chamada *Decomposed Software Pipelining*, onde o problema das dependências de dados e restrição de recursos são divididos em dois subproblemas: (i) livrar o *loop* das depêndencias de dados cíclicos e (ii) livrá-lo da restrição de recursos. Os problemas podem ser tratados em qualquer ordem e experimentos preliminares mostraram que independente da ordem escolhida, os resultados são eficientes e levam a um melhor resultado do que as abordagens existentes tanto em complexidade de tempo quanto em eficiência de espaço com uma menor complexidade de computação [WE, 1993].

[AN, 1988] utiliza uma abordagem onde o programa é representado por seu grafo de dependências. O algoritmo examina uma execução parcial de um *loop,* por exemplo as primeiras *i* iterações, e tenta escalonar estas *i* iterações o mais cedo possível. As dependência entre *loops* são tratadas da seguinte forma: se a maior cadeia de dependências onde a instrução x depende tem comprimento j então x é escalonado no tempo j.

Esta abordagem tenta se aproveitar do fato de que, como as dependências de um *loop* apresentam certas regularidades especificadas pelo grafo de depêndencias, então ao escalonar uma grande porção da execução de um *loop*, algum comportamento repetitivo pode ser revelado. Este fato pode ser usado para obter um escalonamento ótimo para o *loop*.

8.3 Algoritmos Pessimistas

[Pou, 1994] apresenta um método de compactação que é uma generalização do *Percolation Scheduling* [Nic, 1985] e *Perfect Pipelining* [Aik, 1988] e mostra que também, é possível aplicar os métodos de compactação usadas em linguagens imperativas aos programas escritos em linguagens funcionais. O método proposto, também denominado algoritmo pessimista [Ebc, 1987] se caracteriza pela realização de uma execução simbólica de um programa tentando executar as operações o mais cedo possível. Para isto eles utilizam um desmembramento controlado de *loops* até que um padrão repetido ocorra. O programa, nesta abordagem, é representado por um árvore e toda vez que é detectado um nodo pronto, os mesmos são movimentados para cima. Um nodo é dito estar pronto se todos os seus argumentos já foram computados. Ao encontrar um nodo pronto que é equivalente a um já compactado, ele é substituido por uma chamada para uma nova função que executa os mesmos comandos do nodo pronto. Em outras palavras, o processo de compactação, detecta nodos prontos e os passa para cima para que sejam despachados o mais rápido possível fazendo as transformações no programa, como no método *Percolation Scheduling* apresentado na Seção 5.2.1.2. Quando uma chamada recursiva é detectada é feito um desdobramento, como na técnica de *loop unrolling*. E assim sucessivamente até que a condição de parada seja atingida. A condição de parada é a detecção de um nodo pronto igual a outro nodo pronto avaliado anteriormente. Neste momento a chamada para a função recursiva é substituida por uma outra chamada correspondente. Uma função é recursiva quando ela invoca a si mesma direta ou indiretamente.

O método proposto por Pouzet [Pou, 1994] pode ser melhor entendido observando seu exemplo. Neste exemplo, suponha um trecho de programa em uma linguagem

funcional que define uma função *f* que recebe como argumentos uma lista *x* e um valor escalar *accu*. A função *f* executa as primitivas *f1* e *f2* e chama a si mesmo. As barras horizontais indicam os nodos prontos, o par [ ] representa uma lista vazia, *hd* representa a cabeça da lista e *tl* representa a cauda da lista.

$$\begin{array}{l}\text{let rec f} = \lambda x,\ \text{accu. if } x = [\,] \\ \qquad\qquad\qquad\qquad \text{then accu} \\ \qquad\qquad\qquad\qquad \text{else f}\,(\overline{\text{tl(x)}},\ \text{f2}(\ \text{f1}(\overline{\text{hd(x)}}), \text{accu}))\end{array} \qquad \text{(a)}$$

O teste if x = [ ] , e as funções *hd* e *tl* são nodos prontos, portanto devem ser movimentados para cima. O teste não é movimentado pois já se encontra na posição correta. Após a movimentação das funções *hd* e *tl* obtém-se a seguinte configuração:

$$\begin{array}{l}\text{let rec f} = \lambda x, \text{accu. if } x = [\,] \\ \qquad\qquad\qquad\qquad \text{then accu} \\ \qquad\qquad\qquad\qquad \text{else let x1} = \text{tl(x)} \\ \qquad\qquad\qquad\qquad\qquad\quad \text{x2} = \text{hd(x)} \\ \qquad\qquad\qquad\qquad \text{in } \overline{\text{f(x1, f2(f1(x2).accu))}}\end{array} \qquad \text{(b)}$$

Agora o próximo passo é substituir a chamada recursiva de *f* pela sua definição obtendo:

$$\begin{array}{l}\text{let rec f} = \lambda x,\ \text{accu. if } x = [\,] \\ \qquad\qquad\qquad\qquad \text{then accu} \\ \qquad\qquad\qquad\qquad \text{else let x1} = \text{tl(x)} \\ \qquad\qquad\qquad\qquad\qquad\quad \text{x2} = \text{hd(x)} \\ \qquad\qquad\qquad\qquad \text{in} \left(\begin{array}{l} \lambda x,,\ \text{accu. if } \overline{\text{x} = [\,]} \\ \qquad\qquad \text{then accu} \\ \qquad\qquad \text{else let x1} = \text{tl(x)} \\ \qquad\qquad\qquad\quad \text{x2} = \text{hd(x)} \\ \qquad\qquad\quad \text{in f(x1, f2(f1(x2), accu))} \end{array} \right) (\text{x1}, \text{f2}(\overline{\text{f1(x2)}}, \text{accu}))\end{array} \qquad \text{(c)}$$

Agora, movimentando os nodos prontos para cima, x = [ ] e f1(x2), e renomeando o primeiro x do trecho entre parêntesis para x5 e , temos:

```
let rec f = λx, accu. if x = [ ]
                      then accu
                      else let x1 = tl(x)
                               x2 = hd(x)                                    (d)
                           in  let x3 = f1(x2)
                               in if x1 = [ ]
                                  then ( λ accu. accu) (f2(x3,accu))
                                  else ( λ accu. let x5 = tl(x1)
                                                 x2 = hd(x1)
                                                 in f(x5, f2( f1(x2), accu)) ) (f2(x3,accu))
```

Repetindo o processo de mover os nodos prontos e fazendo a redução de *(λ accu.accu)(f2 (x3, accu))* para *f2(x3, accu)* , chegamos a:

```
let rec f = λx, accu. if x = [ ]
                      then accu
                      else let x1 = tl(x)
                               x2 = hd(x)
                           in let x3 = f1 (x2)                               (e)
                              in if x1 = [ ]
                                 then f2 (x3, accu)
                                 else let x4 = f2 (x3, accu)
                                          x5 = tl (x1)
                                          x2 = hd (x1)
                                      in λ accu. (f ( x5, f2 ( f1 (x2), accu)) (x4)
```

O termo λ *accu. f (x5, f2(f1(x2),accu))(x4)* pode ser reduzido a *f(x5, f2(f1(x2),x4))* que já está presente na letra (b), então com esta redução chegamos ao fim da compactação e o código resultante define uma nova função *ff*.

Substituindo assim *f(x5, f2(f1(x2),x4))* por uma chamada para esta função obtem-se o código final que não pode mais ser compactado:

```
let rec ff = λx2, x1, accu. let x3 = f1 (x2)
                            in if x1 = [ ]
                               then f2 (x3, accu)
                               else let x4 = f2 (x3, accu)
                                        x5 = tl(x1)
                                        x2 = hd(x1)                          (f)
                                    in ff (x2, x5, x4)

let rec f = λx, accu. if x = [ ]
                      then accu
                      else let x1 = tl(x)
                               x2 = hd(x)
                           in ff (x2, x1, accu)
```

No exemplo mostrado, a função f é *tail-recursive* (Veja Seção 5.4) e os argumentos são colocados nos registradores assim que possível. Renomeação e operações de

movimentação são condicionadas pelos recursos disponíveis. Nos laços em geral, como foi visto na Seção 5.2.2, a linearização tem que ser controlada e a estratégia onde um pedaço do código é linearizado não converge. O mesmo acontece com funções que possuem mais de uma chamada recursiva. [Pou, 1994] propõe como controle limitar o conjunto de nodos prontos e além disso, ele propõe que quando o conjunto de nodos prontos selecionados de um termo é incluido em um subtermo que já foi compactado, o subtermo é substituido por uma chamada para a nova função definida. Desta forma, o processo de compactação proposto por ele é capaz de compactar, também, funções *non tail-recursive*.

5.4 Análise das Técnicas de Compactação

Fazendo uma análise das duas abordagens examinadas para a compactação global, chegamos à seguinte conclusão: a técnica *Trace Scheduling* [Fisher, 1981] é vantajosa pois é uma técnica de compactação global, que explora paralelismo além dos blocos básicos. Entretanto, apesar de poder ser usada em qualquer aplicação, foi demonstrado na literatura que é mais eficiente para aplicações científicas [FDS, 1992]. Outras desvantagens são: a interação com o usuário não é satisfatória, o paralelismo nos limites dos *traces* pode deixar de ser explorado e o código de reparo incluído nos pontos de entrada e saída do *trace* compactado possui um certo custo.

A técnica *Percolation Scheduling* [Nicolau, 1985], foi desenvolvida com o objetivo de superar as deficiências de *Trace Scheduling* [Fisher, 1981] citadas acima e, portanto apresenta uma maior interação com usuário. Explora o paralelismo além dos *traces* e não é necessária a inclusão de código de reparo incluidos nos pontos de entrada e saída do *trace* compactado. Contudo, a aplicação de *Percolation Scheduling* [Nicolau, 1985] pode ser muito trabalhosa e os rearranjos do grafo resultante das aplicações das primitivas também acarretam um certo custo. Uma forma de obter melhores resultados seria usá-la em conjunto com a técnica *Trace Scheduling* [Fisher, 1981]

Em relação às duas abordagens de compactação de *loops* estudadas, verificou-se que a combinação da técnica *Loop Unrolling* com *software pipelining* [Lam, 1988] pode produzir um melhor resultado, pois *loop unrolling* reduz o código extra, enquanto que o *software pipelining* [Lam, 1988] reduz o custo de *startup* de cada iteração. A Figura 12 da Seção 5.2.2.2 ilustrou este fato por meio de um exemplo. Mas cuidados especiais devem ser tomados porque embora algumas vezes, o uso de *loop unrolling* juntamente com *software pipelining* pode aumentar a eficiência do tempo de execução, esta combinação pode degradar a eficiência em termos de espaço, se o *loop* for desmembrado um número excessivo de vezes.

Outra consideração importante diz respeito às transformações de funções com recursividade de cauda em laços. Uma função é *tail-recursive*, ou com recursividade de cauda, se a última ação da função é chamar a si mesma e retornar o valor da chamada recursiva sem executar qualquer outro comando, isto é, não há computações a serem feitas depois do retorno da chamada recursiva. Este tipo de função pode ser eliminado, traduzindo-a em um laço pela substituição da chamada da função por um *goto* para o seu início. A Figura 13, extraída de [BGS, 1994] ilustra a eliminação da recursividade de cauda e mostra um exemplo de função sem recursividade de cauda.

```
recursive logical function inarray(a, x, i, n)
        real x, a[n]
        integer I, n

        if (i > n) then
            inarray = .FALSE.
        else if (a[i] = x) then
            inarray = .TRUE.
        else
            inarray = inarray(a, x, i+1, n)
        end if
        return
```

(a) Procedimento com recursividade de cauda

```
recursive logical function inarray(a, x, i, n)
        real x, a[n]
        integer I, n

1       if (i > n) then
            inarray = .FALSE.
        else if (a[i] = x) then
            inarray = .TRUE.
        else
            i = i+1
            goto 1
        end if
        return
```

(b) Depois da eliminação da recursividade

```
recursive integer function sumarray(a,x,i,n)
        real x, a[n]
        integer I, n

        if (i=n) then
           sumarray = a[i]
        else
           sumarray = a[i]+sumarray(a, x, I+1, n)
        end if
        return
```

(c) Um procedimento *non-tail recursive*

Figura 13: Eliminação de recursividade de cauda

Dado este fato, uma generalização natural do princípio de *software pipelining* [Lam, 1988] pode ser utilizada para funções recursivas visto que a recursividade das funções *tail-recursive* pode ser eliminada como mostrado.

6 Conclusão

Esta monografia apresentou o resultado do estudo de uma classe de arquiteturas chamada superescalares, de alguns dos inúmeros métodos de compactação existentes para linguagens imperativas e as principais características das linguagens funcionais. Deste estudo chegamos a seguinte conclusão:

As arquiteturas superescalares foram desenvolvidas graças ao estudo de técnicas de exploração do paralelismo existente no interior de um único processador. Uma das principais características das arquiteturas superescalares é a existência de várias unidades funcionais que podem operar em paralelo. A exploração do paralelismo de baixo nível nestas arquiteturas se denomina compactação.

A compactação para as arquiteturas superescalares consiste em reconhecer grupos de instruções elementares para serem escalonadas em paralelo e observando-se dois fatores importantes: a máxima utilização das unidades funcionais e a equivalência semântica do programa. O compilador extrai o paralelismo dos programas a partir de uma análise de dependência de suas instruções. Para que um método de compactação se torne utilizável dentro de todos os compiladores das arquiteturas superescalares é essencial dispor de uma implementação eficiente tanto para a compactação global como para a compactação de *loops* independente do paradigma da linguagem usada.

As linguagens funcionais apresentam várias características que oferecem mais oportunidades de explorar o paralelismo e facilidades inerentes as arquiteturas superescalares do que as linguagens imperativas. Por exemplo, o paralelismo nas linguagens funcionais é natural. O não-determinismo no momento da avaliação é natural. Pela própria natureza do DAG, se houver n unidades funcionais ou n processadores nas arquiteturas paralelas, cada ramo do DAG ou árvore pode ser disparado em paralelo. Portanto em linguagens funcionais aproveita-se melhor o paralelismo já que não existe o conceito de atualização de variáveis nestas linguagens e nem efeito colateral. Em contrapartida existem também alguns fatores que podem degradar a execução de programas funcionais, um deles seria a inexistência de uma arquitetura funcional. Na verdade este é um problema fundamental das linguagens funcionais, porque as máquinas foram construídas para as linguagens imperativas onde existe a noção de atualização de memória o que não é natural em uma linguagem funcional. As arquiteturas de V*on Neumann* não são adequadas para a execução de programas em linguagens funcionais. Mas como não existe uma máquina funcional, uma forma de compactar programas neste novo paradigma é traduzí-la para linguagens imperativas e então aplicar as transformações no grafo de controle de fluxo obtido, o que não é uma boa tática já que assim as propriedades das linguagens funcionais propícias para a compactação desapareceriam na tradução.

No estudo das técnicas de compactação foi possível observar que as técnicas de compactação global oferecem mais oportunidades de explorar o paralelismo do que a compactação local pois exploram o paralelismo existente além dos blocos básicos e que o tratamento dos laços é extremamente importante para que seja alcançado um bom nível de eficiência. Podemos concluir, também, que uma combinação das técnicas apresentadas neste trabalho é possível para aumentar a eficiência. Entretanto, interesses e propósitos devem ser avaliados para que a escolha de combinações de algoritmos seja adequada. Não existe uma técnica melhor do que a outra para todas as classes de programas. O que ocorre é que cada técnica é adequada para um determinado tipo de programa. Por exemplo, a Figura 12 ilustrou que *software pipelining* era mais eficiente que *loop unrolling* para aquele trecho de programa, entretanto em outros trechos o inverso pode ocorrer. O mesmo acontece com *Trace Scheduling* e *Percolation Scheduling*, um método pode ser mais eficiente para determinado caso, e menos eficiente para outros. Se a linguagem usada é uma linguagem funcional a questão da manutenção do grafo de dependências é mais simples porque a reconstrução deste grafo pode ser evitada devido ao fato de não haver atualizações de variáveis durante a execução de um programa funcional. As dependências não evoluem ao longo do tempo, evitando assim o doloroso problema de modificar o grafo várias vezes. Com a construção do grafo de dependências inicial

têm-se uma representação finita e estática do programa, o que facilita bastante o trabalho.

Os métodos de compactação de *loops* e global também podem ser utilizados conjuntamente. Não existe um método de compactação de laços completos que permita tomar partido de todo o paralelismo potencial de um programa, principalmente se o generalizarmos para programas recursivos. Este problema é muito complexo e um dos motivos desta complexidade consiste na ignorância durante a compilação do caminho de execução. O compilador pode compactar trechos que não serão executados ou deixar de compactar eficazmente trechos que serão executados.

A técnica apresentada na Seção 5.3 mostrou-se como uma boa solução para a compactação de linguagens funcionais nas arquiteturas superescalares apesar de ter ainda muitos pontos a serem pesquisados. Seu esquema é válido porque mostrou que é possível, fazendo algumas adaptações ou combinando técnicas propostas para programas escritos em linguagens imperativas aplicá-las em linguagens funcionais. Foi possível observar nesta técnica também a viabilidade de compactação das funções recursivas gerais. As funções *tail-recursive* já eram tratadas pelas técnicas anteriormente desenvolvidas, pois este tipo de função pode ser eliminado, transformando-a em um laço através da substituição da chamada da função por um *goto* para o seu início. Já as funções *non-tail recursive* não podem ser eliminadas tão facilmente. Esta técnica é capaz de compactar este tipo de função também, pois ela apresenta uma condição de parada, o que não existia nas técnicas anteriores. O controle proposto por [Pou, 1994] que consiste na limitação do conjunto de nodos prontos possibilida a compactação deste tipo de função recursiva também.

Contudo, não foi demonstrado pelo autor se realmente existe um ganho em termos de execução. Ele não implementou os algoritmos propostos. O que ele fez foi simplesmente mostrar uma série de transformações aplicadas diretamente a alguns programas fontes escritos em uma linguagem funcional proposta por ele. Como continuação deste trabalho, seria interessante tentar implementar a técnica proposta para compactação em uma linguagem funcional real tendo em vista sua execução em uma máquina real e fazer um estudo de sua execução.

7 Bibliografia

[ACG,1994] Avelar, C. U., Chaimowicz, L, Giacomim, R.. *Processadores Pipelined Avançado* - Seminário de Arquitetura de Computadores - DCC - UFMG, Maio/1994.

[Aik, 1988] Aiken, A. S. *Compaction-based Parallelization.* Ph.D Thesis. Cornell University. 1988.

[ANN, 1996] Aiken, A. ,Nicolau, A, Novack, S. *Resource-Constrained Software Pipelining.* In IEEE Transactions on Parallel and Distributed Systems, 1996.

[AN, 1988] Aiken, A., Nicolau, A. *Optimal Loop Parallelization.* In Procedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. Atlanta, Georgia, June 22-24, 1988.

[Ara, 1986] Arabe,J. N. C. *Compiler Considerations and Run-Time Storage Management for a Functional Programming System.* Ph.D. Thesis. University of California, Los Angeles, 1986

[ASU, 1986] Aho, A. V., Sethi, R., Ullman, J. D. *Compilers Principles, Techniques, and Tools,* Addison Wesley, 1986.

[Aug, 1984] Augustsson, L. 1984 . *A Compiler for lazy ML.* Proceedings of the ACM Symposium on Lisp anf Functional Programming, Austin. August, pp. 218-27.

[BGS,1994] Bacon, D. F., Graham, S. L., Sharp, O. J. *Compiler Transformations for High-Performance Computing.* ACM Computers Surveys, Vol.26, No. 4, December, 1994.

[Bur, 1980] Burstall, R. M., MacQueen, D.B. and Sanella, D. T. 1980. *Hope: an experimental aplicative language.* CSR-62-80. Department of Computer Science, University of Edinburgh. May.

[BW, 1988] Bird, R., Wlader, P. *Introduction to Functional Programming.* C.A.R. HOARE Series Editor, Prentice-Hall, 1988.

[Big, 1994] Bigonha, R. S. Script an an Object Oriented Language for Denotational Semantics (user's manual and reference). RT 03/94, DCC, UFMG, 1994.

[Ebc, 1987] Ebcioglu, K. *A compilation technique for software pipelining of loops with conditional jumps.* In Procedings of the 20th Annual Workshop on Microprograming, pages 69-79, December 1987.

[EN, 1989] Ebcioglu, K., Nicolau, A. *A global resource-constrained parallelizttion technique.* In Conference on Supercomputing, June 5-9 1989.

[EW, 1993] Eisenbeis, C, Windheiser, D. *A new class of algorithms for software pipelining with resources constraints.* Technical report, INRIA, 1993.

[FER, 1984] Fisher, J. A., Ellis, J. R., Ruhenberg, J. C. and Nicolau, A. *Parallel processing: A Smart Compiler and a Dumb Machine.* In Symposium on Compiler Construction. SIGPLAN Notices, June 1984. Vol 19, Number 6.

[Fis, 1981] Fisher, J. A. *Trace Scheduling: A Technique for Global Microcode Compaction.* In IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981.

[FDS, 1992] Fernandes, E. S. T., Santos, A. D. *Arquiteturas Superescalares: Detecção e Exploração de Baixo Nível.* Porto Alegre: Instituto de informática da UFRGS, 1992, 1148p.

[FV,1991] Figueiredo, L. C. e Vecchio, C. C. L. *Arquiteturas VLIW.* Relatório Técnico RT016/91 - DCC - UFMG - 1991.

[Gor, 1979] Gordon, M.J., Milner, A. J. and Wadsworth, C. P. 1979. *Edinburg LCF.* LNCS 78. Springer Verlag.

[GS, 1990] Gupta, R. and Soffa, M. L. *Region Scheduling: An Approach for Detecting and Redistributing Parallelism.* In IEEE Transation on Software Engineering, 16: 421-431, 1990.

[GS, 1992] Gasperoni, F. and Schwiegelshohm, U. *Scheduling loops on parallel processors: A simple algorithm with close to optimum performane.* In CONPAR, September 1992.

[HG, 1982] Hennessy, J. L., Gross, T. R. *Code generation and reorganization in the presence of pipeline constraints.* In Principles of Programming Languages, pages 120-127, 1982.

[HP, 1990] Hennessy, J. L., Patterson, O. A.. *Computer Architecture A Quantative Approach.* Morgan Kaufmann Publishers Inc, 1990.

[Lam, 1988] Lam, M. S. *Software pipelining: An effective scheduling technique for VLIW machines.* In Conference on Programming Language, Design and Implementation, pages 318-328, June 22-24 1988.

[Mai, 1996] M. A. Maia, *Implementação de Linguagens Funcionais*, RT XX/96, DCC/UFMG, 1996.

[MBig, 1994] Bigonha, M. A. S. *Otimização de Código em Máquinas Superescalares.* Tese de Doutorado. Departamento de Informática, PUC - RJ, Abril, 1994.

[MBig, 1994a] Bigonha, M. A. S., Rangel, J. L. *Sistema Gerador de Geradores de Código para Arquiteturas Superescalares.* In Anais do VII SBRC, Canela, RS, Jullho, 1995.

[ME, 1992] Moon, S. M., Ebcioglu, K. *An efficient resource-constrained global scheduling technique for superscalar and VLIW processors.* In MICRO *25*, pages 55-71, 1992.

[Nic, 1985] Nicolau, A. Percolation Scheduling: A Parallel Compilation Technique. Technical Report. Cornell University. 1985.

[Pey, 1987] Peyton, J. S. *The Implementation of Functional Languages*, Prentice-Hall International, 1987.

[Pin, 1988] Pingali, K. *Fine-grain compilation for pipelined machines.* Technical Report 88-934, Cornell University, August 1988.

[Pou, 1994] Pouzet, M. *Fine Grain Parallelisation of Functional Programs for VLIW or Superscalars Architectures.* In International conference on Applications in Parallel and Distributed Computing, April 1994.

[Tur, 1976] Turner, D. A. *The SASL Language Manual.* University of St Andrews. December, 1976.

[Tur, 1985] Turner, D. A. Miranda - a non-stricrt functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture,* Nancy, pp. 1-16. Jouannaud, LNCS 201. Springer Verlag.

[Tur, 1982] Turner, D. A. Recursion Equations as a Programming Language. In *Functional Programming and its Applications,* Darlington *et al.*, pp 1-28. Cambridge University Press.

[Wad, 1985] Wadler, P. 1985. Introduction to Orwell. Programming Research Group. University of Oxford.

[WE, 1993] Wang, J., Eisenbeis, C. *Decomposed software pipelining: a new approach to exploit instruction level parallelism for loop programs.* In IFIP WG 10.3 Working Conference on Architectures and Compilation Techiniques for Fine and Medium Grain Parallelism, January, 1993.

Patrícia Campos Costa

Patrícia Campos Costa

Mariza A. S. Bigonha

Mariza Andrade da Silva Bigonha