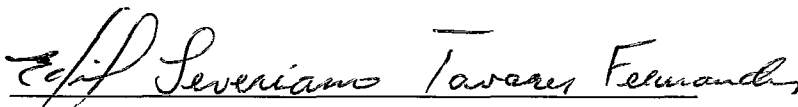


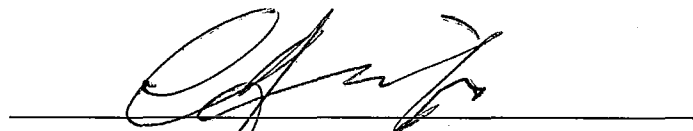
EFEITO DA EXECUÇÃO CONDICIONAL EM ARQUITETURAS PARALELAS


Anna Dolejsi Santos

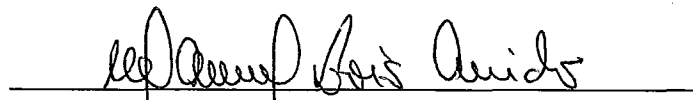
TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS.

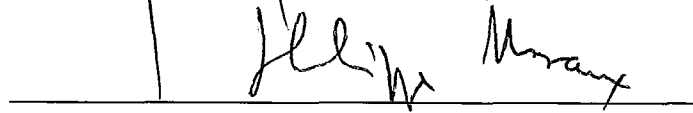
Aprovada por:

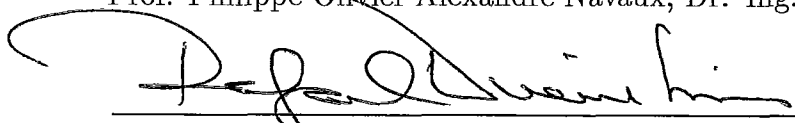

Prof. Edil Severiano Tavares Fernandes, Ph.D.
(Presidente)


Prof. Claudio Luis de Amorim, Ph.D.


Prof. Valmir Carneiro Barbosa, Ph.D.


Prof. Manuel Lois Anido, Ph.D.


Prof. Philippe Olivier Alexandre Navaux, Dr. Ing.


Prof. Rafael Dueire Lins, Ph.D.

SANTOS, ANNA DOLEJSI

Efeito da Execução Condicional em Arquiteturas Paralelas

[Rio de Janeiro] 1994.

XXIV, 171 p., 29.7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas, 1994)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Arquiteturas Super Escalares 2. Paralelismo Baixo Nível

3. Execução Condicional

I. COPPE/UFRJ II. Título (série)

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

Efeito da Execução Condicional em Arquiteturas Paralelas

Anna Dolejsi Santos

Dezembro, 1994

Orientador : Edil Severiano Tavares Fernandes

Programa : Engenharia de Sistemas e Computação

Uma das tendências atuais para aumentar o desempenho de processadores, consiste em introduzir na arquitetura múltiplas unidades funcionais independentes que podem ser ativadas em paralelo, para executar diferentes instruções do mesmo programa de aplicação. Essas máquinas são denominadas Super Escalares e dependendo do nível de implementação do algoritmo responsável pelo escalonamento das instruções que serão executadas durante cada ciclo de processador podem ser classificadas em:

- processadores com escalonamento dinâmico, e
- processadores com escalonamento estático.

No primeiro tipo, o algoritmo de escalonamento é implementado diretamente no *hardware*. No segundo tipo, o escalonamento das instruções que serão ativadas durante cada ciclo de máquina é levado a cabo por um algoritmo implementado pelo *software* de suporte da arquitetura. Desse modo, o algoritmo de escalonamento é realizado previamente, durante a fase de geração de código. Máquinas VLIW (*Very Large Instruction Word*) são exemplos de processadores com algoritmo de escalonamento estático.

Especificamos e simulamos uma nova modalidade de processador Super Escalar VLIW no qual as instruções são executadas condicionalmente. No nosso modelo

de processamento Super Escalar, que denominamos CONDEX, cada instrução de máquina é suficientemente larga para acomodar os campos que controlam as unidades funcionais. Além da operação que será obedecida pela respectiva unidade funcional, cada campo indica a condição que deve ser satisfeita para que a respectiva unidade funcional seja ativada. Desse modo, podemos empacotar em uma mesma instrução larga, instruções provenientes de blocos básicos distintos. Ao empacotar comandos oriundos de blocos básicos distintos, estamos na realidade aumentando o tamanho dos blocos básicos e conseqüentemente o nível de unidades funcionais que podem ser ativadas em paralelo a partir de uma mesma instrução larga.

Desenvolvemos um método para geração de código paralelo executável a partir de código seqüencial para o CONDEX, isto é, uma técnica para escalonamento de instruções. Esse método de escalonamento inclui duas etapas:

- a etapa na qual é realizada a compactação local, e
- a etapa na qual é realizada a compactação condicional.

Durante a compactação local, o programa responsável por essa tarefa, o Compactador Local, recebe como entrada código intermediário seqüencial gerado pelo compilador e inicialmente identifica os blocos básicos do programa e constrói o seu grafo de fluxo de controle. A compactação local propriamente dita, é feita separadamente para cada bloco básico presente no grafo de fluxo do programa. O objetivo é montar uma seqüência de instruções largas paralelas a partir das instruções de cada bloco básico. Para tanto o Compactador Local procura movimentar instruções uma a uma, para cima, tentando grupá-las em instruções. As movimentações levam em conta as dependências de dados e os conflitos na utilização dos recursos da máquina. Essa etapa da geração de código é realizada automaticamente.

A compactação condicional, que atualmente é feita manualmente, não respeita fronteiras de blocos básicos e visa produzir um único bloco a partir das instruções que formam os blocos básicos correspondentes aos blocos THEN e ELSE ou aos blocos THEN e Bloco Sucessor (i.e., o bloco básico para o qual o fluxo de controle é desviado se a condição testada é falsa, ou após a execução do bloco THEN), da linguagem Pascal. Inicialmente são identificados os blocos THEN e ELSE, e os blocos THEN e o Bloco Sucessor, no programa que foi previamente submetido a compactação local. Posteriormente as instruções do bloco ELSE são escalonadas em instruções largas que contêm as instruções do bloco THEN, ou as instruções do Bloco

Sucessor são escalonadas em instruções largas que contêm instruções do THEN. A movimentação das instruções é feita uma a uma, para cima, tentando grupá-las em instruções largas parcialmente preenchidas pelo processo de compactação local. As movimentações levam em conta as dependências de dados, os conflitos na utilização dos recursos da máquina e o tempo de latência das unidades funcionais.

Utilizando uma bateria de programas de teste, realizamos a avaliação da técnica de escalonamento de instruções e o desempenho do modelo de processador proposto. Inicialmente contabilizamos, para cada uma das configurações do processador utilizadas, o efeito estático resultante do emprego da compactação condicional, isto é, verificamos como a compactação condicional reduziu o número de instruções e instruções largas de cada programa teste. O efeito dinâmico da compactação condicional foi medido posteriormente durante interpretação dos programas teste, em cada uma das configurações usadas. Observamos *speedups* no tempo de execução dos programas teste variando na faixa de 1,11 até 1,76.

Finalmente, buscando encontrar métodos alternativos para explorar os recursos presentes no modelo CONDEX, desenvolvemos e avaliamos duas variações do algoritmo de compactação global *trace scheduling* para geração de código paralelo executável a partir de código seqüencial. Nesses experimentos, observamos *speedups* na faixa de 1,13 a 1,84.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

Effect of Conditional Execution in Parallel Architectures

Anna Dolejsi Santos

December, 1994

Thesis Supervisor : Edil Severiano Tavares Fernandes

Department : Programa de Engenharia de Sistemas e Computação

A current trend to improve processor performance consists in incorporating multiple functional units that can be activated to execute in parallel distinct instructions from the same application program. These machines are known as Superscalar, and according to the implementation level of the instruction issue algorithm, they can be classified as:

- dynamic scheduled machines, and
- static scheduled machines.

In the first type, the instruction issue algorithm is embedded in the underlying hardware. In the second type, instruction scheduling is performed by an algorithm implemented by the support software of the architecture. Thus, the instruction scheduling is done previously, during the code generation (or the optimization) phase. VLIW (Very Large Instruction Word) machines are typical examples of processors whose scheduling algorithm is implemented through software.

This work deals with the exploitation of the instruction-level parallelism of superscalar machines with static scheduling algorithms. In this way, we have specified and simulated a new type of VLIW processor in which all the operations are executed conditionally. In our model of Superscalar processor, called CONDEX machine, each very long instruction has enough fields to control the hardware resources.

In addition to the operation which will be obeyed by the corresponding functional unit, each field indicates if the functional unit will be activated during the current processor cycle, based on the bit pattern of the condition code. As a consequence of this conditional execution concept, we can pack in the same long instruction, commands proceeding from different basic blocks. By packing instructions belonging to different basic blocks, we are increasing the length of the basic blocks and so the number of functional units that can be activated on parallel from the same large instruction.

We developed a method to generate parallel code for the CONDEX machine, from the sequential code (i.e., a method for instruction scheduling). This scheduling method is carried out in two steps:

- the local compaction, and
- the conditional compaction step.

During the local compaction step, the program responsible for this task receives as input the intermediary sequential code generated by the compiler. The local compaction algorithm identifies the basic blocks of the program, and constructs the program control-flow graph. The local compaction is done individually for each basic block in the control-flow graph, and its main objective is to build a sequence of very large instructions. Each very large instruction contains the instructions of each basic block. Next, the algorithm tries to move each instruction up, in order to form large instructions. The movement take into account the data dependencies and the existing conflicts in the use of the machine resources. This code generation phase is done automatically.

The conditional compaction process is carried out interactively. The algorithm ignores the basic blocks frontiers and it is responsible for producing a single basic block with the instructions proceeding from the THEN and ELSE blocks or from the THEN and Successor blocks (i.e., the successor block is the basic block to which the control flow is transferred whenever the IF condition is false, or after the execution of the block THEN). The algorithm identifies the pairs of “THEN and ELSE” blocks, and the pairs “THEN and Successor” of the application program. The instructions of the THEN block are placed together with the ELSE block instructions, or together with the instructions of the Successor Block. Instruction movement takes into account the data dependencies, the existing conflicts in the use of machine resources

and the functional unit latency time.

In order to evaluate this instruction scheduling method for some machine configurations derived from our processor model, several experiments were carried out with a set of test programs. Initially, for each machine configuration we observed the static effect of the conditional compaction algorithm, i.e., we analysed the reduction in the number of instructions and the number of very long instructions of each test program. By interpreting the suit of test programs in each machine configuration, we have assessed the impact of the conditional compaction method in the performance of the processors. During these experiments we observed speedup ratios ranging from 1,10 up to 1,75.

Finally, in order to find alternatives methods to use the CONDEX model resources more efficiently, we developed and evaluated two new alternative algorithms which take into account the execution profile of the application programs. These alternative algorithms belong the class of global compaction algorithms. The parallel code generated by these algorithms were interpreted by the CONDEX model simulator. During these experiments, we observed speedup ratios ranging from 1,17 up to 1,84.

Ao meu pai, Odolen Dolejsi, com imensa saudade.
Ao meu filho, Jorge Boccanera, com imensa alegria.

Agradecimentos

Ao meu orientador Edil, pelo permanente incentivo, pela contínua atenção, pela tranqüila sabedoria demonstrada ao longo do desenvolvimento dessa tese e particularmente, por iniciar-me no estudo da detecção e exploração do paralelismo de baixo nível.

Aos professores Claudio Luis de Amorim e Valmir Carneiro Barbosa, que juntamente com meu orientador, possibilitaram a realização do meu exame de qualificação.

Ao professor Nelson Quilula Vasconcelos pelo compartilhamento fraterno do seu exíguo espaço físico.

A Leonardo Cardoso Monteiro por ceder e alterar seu compilador e seu compactador para ser usado nessa pesquisa.

A Felipe Maia Galvão França e Priscila Machado Vieira Lima pela generosa acolhida.

A Lúcia Maria de Assumpção Drummond, incansável companheira das muitas e longas horas de estudos.

A Suely, Denise, Cláudia, Ana Paula e a todos os funcionários da secretaria e do laboratório do Programa de Engenharia de Sistemas e Computação pela solicitude e competência sempre demonstradas.

Ao Departamento de Computação da Universidade Federal Fluminense por ter criado as condições necessárias para a elaboração desse trabalho.

A CAPES e a FAPERJ pelo apoio individual concedido.

A DEUS que renova em mim a cada dia, sonhos, ideais e esperanças.

Índice

1	Introdução	1
1.1	Motivação	1
1.2	Objetivos do Trabalho	5
1.3	Organização do Texto	7
2	Conceitos Básicos	9
2.1	Introdução	9
2.2	Arquiteturas Super Escalares	10
2.3	Técnicas de Detecção de Paralelismo	18
2.3.1	Dependência de Dados e Restrições de Recursos	19
2.3.2	A Compactação	25
2.4	Latências e Número de Instruções Despachadas por Ciclo	31
3	O Modelo	37
3.1	Introdução	37
3.2	O Código Intermediário	37
3.3	A Execução Condicional	43
3.4	O Modelo de Arquitetura	45
3.5	Os Elementos do Modelo	47

3.6	A Conexão Entre os Elementos do Modelo	51
3.7	O Simulador do Modelo	52
4	Compactação de Código	54
4.1	Introdução	54
4.2	A Geração do Código Paralelo	54
4.2.1	O Processo de Compactação Local	56
4.2.2	O Processo de Compactação Condicional	58
5	Avaliação de Desempenho	71
5.1	Introdução	71
5.2	A Metodologia	71
5.3	As Configurações da Arquitetura	72
5.4	Os Programas Teste	72
5.5	Avaliando a Qualidade do Código	73
5.5.1	O Programa “Livermore Loop”	74
5.5.2	O Programa “Branch”	78
5.5.3	O Programa “BCD”	81
5.5.4	O Programa “Simpson”	85
5.5.5	O Programa “Árvore”	89
5.6	Sumário dos Resultados Obtidos	93
6	Técnicas de Escalonamento mais Agressivas	99
6.1	Introdução	99
6.2	Técnicas de Compactação Baseadas no Perfil de Execução dos Programas	99
6.3	Os Experimentos	101

6.3.1	O Programa “Livermore Loop”	101
6.3.2	O Programa “Branch”	104
6.3.3	O Programa “BCD”	107
6.3.4	O Programa “Simpson”	110
6.3.5	O Programa “Árvore”	114
6.4	Sumário dos Resultados Obtidos	117
7	Conclusões	120
7.1	Resumo do Trabalho	120
7.2	Sugestões para Pesquisas Futuras	122
	Bibliografia	124
	A Programas de Teste	133
	B A Utilização das Unidades Funcionais	140

Lista de Figuras

2.1	A Instrução Multifuncional (IMF) de uma Arquitetura VLIW	13
2.2	Um Campo da Instrução Multifuncional de uma VLIW	14
2.3	Componentes de uma Arquitetura VLIW	15
2.4	Esquema de Interconexão dos Módulos de uma Arquitetura VLIW . . .	16
2.5	Instrução Longa de uma Arquitetura VLIW	22
2.6	Blocos Básicos de um Programa	23
2.7	Grafo do Fluxo de Controle de um Programa	29
2.8	Blocos Básicos de um Trecho de Programa	29
2.9	Temporização na Execução Seqüencial com Latências Iguais	33
2.10	Instruções para a Arquitetura Hipotética	33
2.11	Temporização na Execução Seqüencial com Latências Diferentes . . .	34
2.12	Ativação de Múltiplas Instruções por Ciclo com Latências Diferentes .	35
3.1	Um Trecho do Programa <i>Branch</i>	44
3.2	Um Trecho do Programa <i>BCD</i>	44
3.3	Esquema da Arquitetura Modelo – CONDEX	46
3.4	A Seqüência de Instruções para a ALU_i	47
3.5	Um Conjunto de Indicadores de Condição	48
3.6	Efeito da Execução de ICMP R4, R5, 1, se $R4 = 5$ e $R5 = 5$	49

3.7	Efeito da Execução de ICMP R4, R5, 1, se $R4 = 3$ e $R5 = 5$	49
3.8	Efeito da Execução de ICMP R4, R5, 1, se $R4 = 5$ e $R5 = 3$	49
3.9	Registrador de Instrução da Arquitetura Condicional	50
3.10	Um Campo do Registrador de Instrução	50
3.11	A Conexão de ALUs Através do Banco de Registradores	52
4.1	Identificação dos Pares “bt-be” e “bt-bs”	58
4.2	Segunda Fase do Algoritmo – Fusão dos Pares “bt-be”	59
4.3	Terceira Fase do Algoritmo – Fusão dos Pares “bt-bs”	60
4.4	Quarta Fase do Algoritmo – Atualização dos Endereços	61
4.5	Trecho do Programa <i>Livermore Loop 24</i>	62
4.6	GFC do Trecho do Programa <i>Livermore Loop 24</i>	62
4.7	Trecho do Programa <i>Livermore Loop 24</i> após Compactação Local	63
4.8	Trecho do Programa <i>Livermore Loop 24</i> após Compactação Condicional	64
4.9	Trecho do Programa <i>Branch</i>	65
4.10	GFC do Trecho do Programa <i>Branch</i>	65
4.11	Trecho do Programa <i>Branch</i> após Compactação Local	66
4.12	Trecho do Programa <i>Branch</i> após Compactação Condicional	67
4.13	Trecho do Programa <i>Árvore</i>	68
4.14	GFC do Trecho do Programa <i>Árvore</i>	68
4.15	Trecho do Programa <i>Árvore</i> após Compactação Local	69
4.16	Trecho do Programa <i>Árvore</i> após Compactação Condicional	70
5.1	O Grafo de Fluxo de Controle do Programa <i>Livermore Loop</i>	75
5.2	Execução Seqüencial × Execuções com Compactação Local e Condicional do Programa <i>Livermore</i>	78

5.3	O Grafo de Fluxo de Controle do Programa <i>Branch</i>	79
5.4	Execução Seqüencial × Execuções com Compactação Local e Condi- cional do Programa <i>Branch</i>	81
5.5	O Grafo de Fluxo de Controle do Programa <i>BCD</i>	82
5.6	Execução Seqüencial × Execuções com Compactação Local e Condi- cional do Programa <i>BCD</i>	84
5.7	O Grafo de Fluxo de Controle do Programa <i>Simpson</i>	85
5.8	O Grafo de Fluxo de Controle do Procedimento <i>Simp</i> do Programa <i>Simpson</i>	86
5.9	O Grafo de Fluxo de Controle do Procedimento <i>Fun</i> do Programa <i>Simpson</i>	86
5.10	Execução Seqüencial × Execuções com Compactação Local e Condi- cional do Programa <i>Simpson</i>	88
5.11	O Grafo de Fluxo de Controle do Programa <i>Árvore</i>	89
5.12	O Grafo de Fluxo de Controle do Procedimento <i>Inser</i> do Programa <i>Árvore</i>	90
5.13	Execução Seqüencial × Execuções com Compactação Local e Condi- cional do Programa <i>Árvore</i>	92
6.1	<i>Trace</i> do Programa <i>Livermore Loop 24</i>	102
6.2	Execução Seqüencial × Execução TPE e TPE+ do Programa <i>Liver- more Loop</i>	104
6.3	<i>Trace</i> do Programa <i>Branch</i>	105
6.4	Execução Seqüencial × Execução TPE e TPE+ do Programa <i>Branch</i>	107
6.5	<i>Trace</i> do Programa <i>BCD</i>	108
6.6	Execução Seqüencial × Execução TPE e TPE+ do Programa <i>BCD</i>	110
6.7	<i>Trace</i> do Programa <i>Simpson</i>	111
6.8	<i>Traces</i> do Procedimento <i>Simp</i> do Programa <i>Simpson</i>	112

6.9	Execução Seqüencial × Execução TPE e TPE+ do Programa <i>Simpson</i>	113
6.10	<i>Traces</i> do Procedimento <i>Inser</i> do Programa <i>Árvore</i>	115
6.11	Execução Seqüencial × Execução TPE e TPE+ do Programa <i>Árvore</i>	117
A.1	O Programa <i>Branch</i>	133
A.2	O Programa <i>Livemore</i>	134
A.3	O Programa <i>BCD</i>	135
A.4	O Programa <i>Simpson</i>	137
A.5	O Programa <i>Árvore</i>	139
B.1	UFs do tipo MEM Concorrentemente Ativas – Configuração 3	143
B.2	UFs do tipo ALU Concorrentemente Ativas – Configuração 3	144
B.3	UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa <i>BCD</i>	145
B.4	UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa <i>BCD</i>	146
B.5	UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa <i>BCD</i>	146
B.6	UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa <i>BCD</i>	147
B.7	Efeito das Unidades MEMs no Desempenho – <i>Branch</i>	151
B.8	UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa <i>Branch</i>	152
B.9	UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa <i>Branch</i>	152
B.10	UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa <i>Branch</i>	153

B.11 UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa <i>Branch</i>	154
B.12 UFs do tipo FPU Concorrentemente Ativas – Configuração 2	157
B.13 UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa <i>Simpson</i>	158
B.14 UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa <i>Simpson</i>	159
B.15 UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa <i>Simpson</i>	159
B.16 UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa <i>Simpson</i>	160
B.17 UFs do tipo MEM Concorrentemente Ativas–Configuração 2	163
B.18 UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa <i>Livermore Loop</i>	164
B.19 UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa <i>Livermore Loop</i>	165
B.20 UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa <i>Livermore Loop</i>	165
B.21 UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa <i>Livermore Loop</i>	166
B.22 UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa <i>Árvore</i>	169
B.23 UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa <i>Árvore</i>	169
B.24 UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa <i>Árvore</i>	170
B.25 UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa <i>Árvore</i>	171

Lista de Tabelas

2.1	Exemplos de Dependência de Dados	20
5.1	As Configurações da Arquitetura do Modelo CONDEX	72
5.2	Características dos Programas Teste	73
5.3	Efeito da Compactação Local e Condicional no Programa <i>Livermore Loop</i>	75
5.4	Efeito da Compactação Local e Condicional na Execução do Programa <i>Livermore Loop</i>	77
5.5	Efeito da Compactação Local e Condicional no Programa <i>Branch</i>	80
5.6	Efeito da Compactação Local e Condicional na Execução do Programa <i>Branch</i>	80
5.7	Efeito da Compactação Local e Condicional no Programa <i>BCD</i>	83
5.8	Efeito da Compactação Local e Condicional na Execução do Programa <i>BCD</i>	83
5.9	Efeito da Compactação Local e Condicional no Programa <i>Simpson</i>	87
5.10	Efeito da Compactação Local e Condicional na Execução do Programa <i>Simpson</i>	88
5.11	Efeito da Compactação Local e Condicional no Programa <i>Árvore</i>	91
5.12	Efeito da Compactação Local e Condicional na Execução do Programa <i>Árvore</i>	92
5.13	N ^{os} de IMFs e de Instruções Obtidos para a Execução Seqüencial	93

5.14	N ^{os} de IMFs e de Instruções Obtidos por meio da Compactação Local para a Configuração 1	93
5.15	N ^{os} de IMFs e de Instruções Obtidos por meio da Compactação Condicional para a Configuração 1	94
5.16	N ^{os} de IMFs e de Instruções Obtidos por meio da Compactação Local para a Configuração 2	94
5.17	N ^{os} de IMFs e de Instruções Obtidos por meio da Compactação Condicional para a Configuração 2	94
5.18	N ^{os} de IMFs e de Instruções Obtidos por meio da Compactação Local para a Configuração 3	95
5.19	N ^{os} de IMFs e de Instruções Obtidos por meio da Compactação Condicional para a Configuração 3	95
5.20	N ^{os} de IMFs e de Instruções Executadas Sequencialmente	95
5.21	N ^{os} de IMFs e Instruções Compactadas Localmente Executadas na Configuração 1	96
5.22	N ^{os} de IMFs e Instruções Compactadas Condicionalmente Executadas na Configuração 1	96
5.23	N ^{os} de IMFs e Instruções Compactadas Localmente Executadas na Configuração 2	96
5.24	N ^{os} de IMFs e Instruções Compactadas Condicionalmente Executadas na Configuração 2	97
5.25	N ^{os} de IMFs e Instruções Compactadas Localmente Executadas na Configuração 3	97
5.26	N ^{os} de IMFs e Instruções Compactadas Condicionalmente Executadas na Configuração 3	97
6.1	Efeito do TPE e do TPE+ no Programa <i>Livermore Loop</i>	102
6.2	Efeito do TPE e do TPE+ na Execução do Programa <i>Livermore Loop</i>	103
6.3	Efeito do TPE e do TPE+ no Programa <i>Branch</i>	106

6.4	Efeito do TPE e do TPE+ na Execução do Programa <i>Branch</i>	106
6.5	Efeito do TPE e do TPE+ no Programa <i>BCD</i>	109
6.6	Efeito do TPE e do TPE+ na Execução do Programa <i>BCD</i>	109
6.7	Efeito do TPE e do TPE+ no Programa <i>Simpson</i>	112
6.8	Efeito do TPE e do TPE+ na Execução do Programa <i>Simpson</i>	113
6.9	Efeito do TPE e do TPE+ no Programa <i>Árvore</i>	116
6.10	Efeito do TPE e do TPE+ na Execução do Programa <i>Árvore</i>	116
6.11	N ^{os} de IMFs e de Instruções Obtidos para a Configuração 1	117
6.12	N ^{os} de IMFs e de Instruções Obtidos para a Configuração 2	118
6.13	N ^{os} de IMFs e de Instruções Obtidos para a Configuração 3	118
6.14	N ^{os} de IMFs e Instruções Executadas na Configuração 1	118
6.15	N ^{os} de IMFs e Instruções Executadas na Configuração 2	119
6.16	N ^{os} de IMFs e Instruções Executadas na Configuração 3	119
B.1	UFs Ativas do Programa <i>BCD</i> – Execução Seqüencial	140
B.2	UFs Ativas do Programa <i>BCD</i> – Configuração 1	141
B.3	UFs Ativas do Programa <i>BCD</i> – Configuração 2	141
B.4	UFs Ativas do Programa <i>BCD</i> – Configuração 3	143
B.5	UFs Ativas do Programa <i>Branch</i> – Execução Seqüencial	148
B.6	UFs Ativas do Programa <i>Branch</i> – Configuração 1	148
B.7	UFs Ativas do Programa <i>Branch</i> – Configuração 2	149
B.8	UFs Ativas do Programa <i>Branch</i> – Configuração 3	150
B.9	UFs Ativas do Programa <i>Simpson</i> – Execução Seqüencial	155
B.10	UFs Ativas do Programa <i>Simpson</i> – Configuração 1	155
B.11	UFs Ativas do Programa <i>Simpson</i> – Configuração 2	156

B.12 UFs Ativas do Programa <i>Simpson</i> – Configuração 3	157
B.13 UFs Ativas do Programa <i>Livermore Loop</i> – Execução Seqüencial . . .	161
B.14 UFs Ativas do Programa <i>Livermore Loop</i> – Configuração 1	161
B.15 UFs Ativas do Programa <i>Livermore Loop</i> – Configuração 2	162
B.16 UFs Ativas do Programa <i>Livermore Loop</i> – Configuração 3	163
B.17 UFs Ativas do Programa <i>Árvore</i> – Execução Seqüencial	166
B.18 UFs Ativas do Programa <i>Árvore</i> – Configuração 1	167
B.19 UFs Ativas do Programa <i>Árvore</i> – Configuração 2	167
B.20 UFs Ativas do Programa <i>Árvore</i> – Configuração 3	168

Lista de Abreviaturas

ALU	unidade aritmética e lógica para inteiros
bb	bloco básico
be	bloco ELSE
BRANCH UNIT	unidade de processamento de desvios
bs	bloco sucessor
bt	bloco THEN
CONDEX	modelo VLIW com capacidade de EXecução CONdicional
DDG	<i>Data Dependence Graph</i>
FCFS	<i>first come first served</i>
FIFO	<i>first-in-first-out</i>
FPU	unidade para aritmética em ponto flutuante
GFC	Grafo do Fluxo de Controle
IMF	instrução multifuncional ou instrução longa
MEM	unidade de acesso à memória
MIMD	<i>Multiple-Instruction, Multiple-Data stream</i>
NOP	<i>no operate</i>
OPCODE	código de operação
PC	apontador de instrução
SIMD	<i>Single-Instruction stream, Multiple-Data stream</i>

TPE	Técnica baseada no Perfil de Execução
TPE+	Técnica baseada no Perfil de Execução+
UF	Unidade Funcional
VLIW	<i>Very Large Instruction Word</i>

Capítulo 1

Introdução

1.1 Motivação

Arquiteturas Super Escalares, uma recente tendência no projeto de processadores de alto desempenho, são capazes de executar simultaneamente, durante cada ciclo de máquina, diversas instruções procedentes de um mesmo programa de aplicação. Através da exploração desse paralelismo de baixo nível (i.e., paralelismo a nível de instrução de máquina) existente nos programas de aplicação, o tempo de processamento desses programas pode ser reduzido substancialmente num processador Super Escalar. Os processadores RS/6000 da IBM ([BAKO90], [GROH90] e [HEST90]), os modelos i860 ([INTE89] e [KOHN89]) e i960 da Intel ([HINT89], [INTE89a] e [MCGE90]), o MC88110 da Motorola ([DIEF92] e [SMOT93]) e a família Alpha da DEC ([DEC92] e [SITE93]), são exemplos de arquiteturas Super Escalares.

A equipe de Arquitetura de Computadores da COPPE/UFRJ vem desenvolvendo projetos de pesquisa para explorar o paralelismo de baixo nível. Os resultados dessas pesquisas estão relatados em [BORN91], [FERN92], [FERN92a], [FERN92b], [BARB93] e [HSIN93].

Bornstein e Pereira ([BORN91] e [FERN92b]) implementaram as primitivas da técnica de escalonamento de instruções *Percolation Scheduling* ([NICO85]), gerando código paralelo, posteriormente interpretado em um simulador parametrizável, de um modelo de processador VLIW.

Em [FERN92] é possível obter-se uma visão global a respeito das arquiteturas Super Escalares e da detecção e exploração do paralelismo de baixo nível.

Empregando um modelo de máquina Super Escalar, derivado do processador i860 da INTEL, Barbosa avaliou o efeito de detalhes arquiteturais no desempenho de algoritmos de escalonamento dinâmico associativo ([FERN92a] e [BARB93]).

Finalmente Hsing ([HSIN93]) estudou o efeito da predição de desvios e da interrupção precisa no desempenho de processadores Super Escalares.

Processadores Super Escalares são caracterizados pela existência de múltiplas unidades funcionais independentes que podem ser ativadas simultaneamente durante cada ciclo de máquina, viabilizando dessa forma, a execução em paralelo de múltiplas operações [TABA91].

Contribuições clássicas que levaram ao desenvolvimento dos processadores Super Escalares, incluem o projeto do CDC-6600 da Control Data ([THOR64]), do IBM-360/91 ([ANDE67] e [TOMA67]), a especificação do processador HPS ([PATT85]) etc. Por outro lado, como consequência do desenvolvimento de técnicas de compactação global ([TOKO78], [FISH81] e [NICO85]) de micro-programas, surgiram as arquiteturas Super Escalares *Very Large Instruction Word* (VLIW) [FISH84a].

Para que o paralelismo potencial de um processador Super Escalar possa ser efetivamente explorado, torna-se necessário detectar e selecionar as instruções do programa de aplicação que podem ser executadas durante cada ciclo de máquina.

O algoritmo de escalonamento pode modificar a ordem em que as instruções serão executadas, determinando quais as unidades que devem ser ativadas durante cada ciclo de máquina. Esse algoritmo pode ser implementado no *hardware* ou via *software*.

No primeiro caso, o algoritmo é implementado diretamente na unidade de controle do processador Super Escalar, e a detecção do paralelismo de baixo nível é realizada em tempo de execução do programa de aplicação. Por esse motivo, o algoritmo de escalonamento é denominado dinâmico.

Com o objetivo de reduzir a complexidade da unidade de controle, em alguns processadores, a tarefa de escalonamento é realizada durante a fase de geração de código objeto, precedendo o processamento do programa de aplicação, através do *software* de suporte da máquina Super Escalar. Nesse caso, o algoritmo de escalonamento é denominado estático.

É importante observar que podemos combinar esses dois tipos de algoritmos. Em outras palavras, a detecção e exploração do paralelismo de baixo nível é executada pelo algoritmo implementado no *hardware* (como no primeiro caso), mas as instruções dos programas de aplicação podem ser reorganizadas previamente por um algoritmo de escalonamento estático que leva em consideração algumas características do processador Super Escalar. O número e mistura de suas unidades funcionais, o tempo de latência das unidades funcionais e as relações de dependências entre as instruções, são exemplos de características que influenciam o processo de reorganização das instruções.

Dentre os trabalhos descrevendo algoritmos de escalonamento dinâmico, podemos destacar [TJAD70], [TJAD73], [KELL75], [AUTH85] e [ACOS86]. Artigos apresentando algoritmos de escalonamento estático podem ser encontrado em [TOKO78], [FISH81] e [NICO85]. Estudos avaliando o desempenho de diversos algoritmos estão relatados em [WEIS84], [PLES88], [CHAN91] e [BULT91]. Um *survey* abordando a evolução do seqüenciamento de instruções é apresentado em [KRIC91].

O principal objetivo dos algoritmos de escalonamento de instruções, é manter as unidades funcionais do processador Super Escalar permanentemente ocupadas. O conflito no uso dos recursos do *hardware* e as dependências de dados entre as instruções são exemplos de fatores que limitam a taxa de ocupação das unidades funcionais, degradando o desempenho do processador Super Escalar. A inclusão de múltiplas cópias dos diversos tipos de unidades funcionais no processador, não garante a redução do tempo de processamento do mesmo fator [FERN92a]. As dependências de dados por exemplo, impossibilitam a execução antecipada de uma instrução cujo início está condicionado ao término de uma outra que a precede, impedindo desse modo, a ativação de uma unidade funcional ociosa.

Para aumentar o desempenho do processador Super Escalar o algoritmo de escalonamento tenta minimizar o efeito das dependências de dados, retardando o início de instruções dependentes, e antecipando o início de outras. Para que isso seja viável é preciso que o processador possa despachar para as unidades funcionais, instruções fora da ordem especificada originalmente.

Para detectar as instruções que podem ser executadas em paralelo, os algoritmos de escalonamento dinâmico precisam ter acesso a um bloco de instruções de máquina. Por essa razão, esses processadores geralmente incluem uma *janela de instruções* (*instruction buffer*) em seu interior, usada para armazenar instruções

pré-buscadas na memória. Na técnica de despacho sequencial as instruções contidas na janela são atendidas segundo a política FIFO *first-in-first-out*. Ou seja, em cada ciclo (e não ocorrendo conflitos) a instrução mais antiga da janela é examinada, sendo despachada para uma das unidades funcionais. A medida que as instruções da janela são despachadas, outros comandos são trazidos para a janela. Quando não for possível despachar a instrução mais antiga da janela, em razão das dependências de dados ou por falta de uma unidade funcional, o despacho é interrompido e as instruções subseqüentes deixam de ser examinadas, passando a aguardar o despacho da instrução mais antiga contida na janela (i.e., da instrução que provocou a interrupção do mecanismo de despacho).

O desejo de aumentar a ocupação das unidades funcionais existentes na arquitetura, exige que cada instrução seja despachada e executada, tão logo esteja livre de dependências de dados. Portanto, para que esse objetivo possa ser atingido, é necessário permitir não somente a execução fora de ordem (i.e., uma ordem de atendimento diferente da política FIFO), mas também é preciso dotar o processador da capacidade de realizar o despacho de todas as instruções que não apresentam conflito no uso de dados ou de recursos. Ou seja, é necessário realizar o despacho de instruções fora de ordem e de mais de uma instrução por ciclo de processador.

Nos algoritmos de escalonamento estáticos, o início de execução de cada instrução é determinado durante a fase de otimização do código. Durante essa fase, as instruções do programa de aplicação são grupadas em instruções largas (ou muito largas) e para tanto, a ordem originalmente especificada pelo programador é modificada. Quando do escalonamento das instruções, o algoritmo leva em conta as dependências de dados e as restrições no uso dos recursos da arquitetura.

Com a finalidade de promover um melhor escalonamento de instruções em árvores de decisão, a execução condicional (ou execução guardada) em processadores escalares foi proposta por Hsu e Davidson ([HSU86] e [HSU86a]). Para atingir esse objetivo, os autores especificaram um processador cujas instruções de desvio incondicionais e *stores* são guardadas (*guarded instructions*). Essas instruções guardadas permitem transferir o fluxo de controle para o ramo correto da árvore de decisão, permitindo o preenchimento daqueles *slots* de tempo ociosos, resultantes da técnica *delayed branch* [GROS82].

Recentemente, a equipe da Universidade Hertfordshire apresentou resultados sobre o efeito da execução condicional em uma arquitetura VLIW ([STEV94] e

[GRAY94]). A execução guardada nesse caso, proporcionou uma taxa de aceleração (*speedup*) de até 1.76 durante a execução de programas não numéricos.

A execução condicional é um tópico que tem sido investigado em diversas Universidades americanas e fabricantes de processadores. Por exemplo, na Universidade Wisconsin-Madison, Sohi e Pnevmatikatos [PNEV94] apresentam um estudo do efeito obtido ao associar o conceito de execução condicional com técnicas de predição de desvios em arquiteturas Super Escalares com algoritmo de escalonamento dinâmico.

1.2 Objetivos do Trabalho

Durante o processamento de um programa, verifica-se que a freqüência de execução das instruções de desvios é bastante elevada. Instruções desse tipo delimitam estruturas denominadas *blocos básicos*: conjunto de instruções do fluxo de controle do programa de aplicação que será integralmente executado toda vez que o controle for transferido para o início do bloco básico. O número de instruções constituindo um bloco básico geralmente é muito pequeno: em média cinco instruções ([SHUS77], [GROS82], [MCFA86] e [DAVI90]). Tendo em vista que a condição usada pela instrução de desvio pode estar sendo avaliada, então o algoritmo de escalonamento precisa interromper o despacho de instruções até que a nova direção no fluxo de controle seja determinada. Durante essa interrupção, as unidades funcionais permanecerão ociosas, resultando numa queda no desempenho do processador Super Escalar.

Mecanismos de predição de desvios ([SMIT81], [JLEE84], [LILJ88] e [BUTL91]), execução especulativa de um dos blocos básicos sucedendo um desvio condicional e a correspondente inclusão de código reparando os efeitos produzidos pela execução especulativa ([HEST90] e [TABA91]), são exemplos de artifícios explorados pelas arquiteturas Super Escalares da atualidade .

Ao invés de simplesmente reproduzir o *estado da arte* nessa nova modalidade de processamento de alto desempenho, preferimos abordar o problema da queda de desempenho em processadores Super Escalares provocada pelos comandos de desvio condicional sob uma nova perspectiva. Para tanto:

- Especificamos um novo modelo de processador Super Escalar (que deno-

minamos CONDEX);

- Implementamos o correspondente interpretador;
- Desenvolvemos um algoritmo para geração de código paralelo para o modelo de arquitetura;
- Realizamos a avaliação do efeito do algoritmo no desempenho do modelo;
- Desenvolvemos e avaliamos dois algoritmos alternativos para geração de código paralelo para o modelo de arquitetura. Da mesma forma como ocorre com o algoritmo de compactação global *trace scheduling*, esses algoritmos alternativos são baseados no perfil de execução dos programas

Na nova modalidade de processador Super Escalar que especificamos, todas as instruções são executadas condicionalmente. Como num processador VLIW, nesse nosso modelo de processamento, cada instrução de máquina é suficientemente larga para acomodar os campos que controlam as unidades funcionais. Além da operação que será obedecida pela respectiva unidade funcional, cada campo inclui a condição que deve ser satisfeita para que a respectiva unidade funcional seja ativada. Desse modo, podemos empacotar em uma mesma instrução larga, instruções provenientes de blocos básicos distintos, tornando-se desnecessário introduzir no projeto da máquina Super Escalar, mecanismos de predição de desvio e código de reparo.

O objetivo da simulação do modelo é avaliar a qualidade do código paralelo que foi gerado pelos nossos algoritmos de escalonamento a partir do código seqüencial. Adicionalmente, através da interpretação do código, podemos observar o efeito da execução condicional e coletar informações essenciais para o projeto de configurações derivadas do modelo de arquitetura, apresentando uma relação “custo \times benefício” favorável. Para tal, o simulador foi implementado de modo a permitir, através da variação de seus parâmetros, a reprodução do comportamento de uma classe de processadores derivados do modelo básico. Nessa classe, uma configuração difere de uma outra pelo número de unidades funcionais, pelo tempo de latência de suas operações etc.

O processo de geração código paralelo para o modelo de arquitetura proposto, compreende duas fases distintas. Na primeira delas, cada bloco básico do código seqüencial é submetido individualmente a um processo de compactação local. Na segunda fase os blocos básicos previamente compactados, são submetidos (quando

possível) à compactação condicional. Ao empacotar (na segunda fase) comandos pertencentes a blocos básicos distintos, estamos na realidade aumentando o tamanho dos blocos básicos e conseqüentemente o nível de unidades funcionais que podem ser ativadas em paralelo a partir de uma mesma instrução longa. A primeira fase do processo de geração de código paralelo é realizada automaticamente, e a segunda é feita interativamente.

A avaliação da qualidade do código paralelizado pelos nossos algoritmos de escalonamento, foi levada a cabo, no simulador parametrizado do modelo CONDEX. Uma bateria de programas de teste, previamente selecionados e compactados pelas duas fases do processo de geração de código paralelo, foi utilizada nessa avaliação.

Finalmente, visando encontrar métodos alternativos para explorar mais efetivamente os recursos das diversas configurações derivadas do modelo, realizamos um conjunto de experimentos adicionais. Durante esses experimentos, utilizando a mesma bateria de programas de teste, geramos (interativamente) código executável empregando duas novas modalidades de algoritmo de compactação global: algoritmos baseados no perfil de execução de cada programa de teste. A interpretação dos componentes da bateria de teste permitiu avaliar essas duas novas políticas de escalonamento.

1.3 Organização do Texto

Esse trabalho está organizado em sete capítulos e dois apêndices. Iniciamos o Capítulo 2 com uma breve apresentação das arquiteturas Super Escalares, mostrando suas principais características. Apresentamos os dois tipos de algoritmos de escalonamento de instruções dessas máquinas e discorremos sobre os tipos de conflitos que são considerados durante o escalonamento de instruções. Finalizamos o Capítulo 2 mostrando a vantagem promovida pela adoção de tempos de latência distintos para diferentes instruções, no desempenho do processador.

No Capítulo 3 descrevemos o modelo de processador Super Escalar cujas operações são executadas condicionalmente. A descrição inclui a definição dos elementos do processador proposto e como eles são interconectados. O capítulo também apresenta as principais características do simulador parametrizado que implementamos.

O Capítulo 4 descreve as duas fases do processo de escalonamento de instruções. Típicos exemplos de trechos de programas processados pelo algoritmo de compactação condicional, complementam o capítulo.

No Capítulo 5 apresentamos o meio ambiente de avaliação do modelo de arquitetura proposto. A metodologia dos experimentos, a descrição da bateria de programas de teste, o efeito da compactação no tamanho do código e o tempo consumido durante a interpretação de cada programa de teste, são tópicos abordados no capítulo.

No Capítulo 6 descrevemos dois algoritmos de compactação global alternativos. Quando da geração do código paralelo, esses algoritmos utilizam o perfil da execução do programa de teste para decidir quais os blocos básicos que serão intercalados. Mostramos também nesse capítulo, o impacto produzido por esses dois algoritmos no tamanho dos componentes da bateria de teste e no tempo de processamento, requerido pelas diversas configurações do modelo CONDEX.

O corpo principal do texto é encerrado no Capítulo 7 com as conclusões e sugestões para possíveis trabalhos futuros. Os Apêndices A e B contêm respectivamente, a listagem dos programas da bateria e o nível de utilização das unidades funcionais das diferentes configurações.

Capítulo 2

Conceitos Básicos

2.1 Introdução

A incansável busca por máquinas mais rápidas, em conjunto com os avanços na tecnologia de produção de circuitos integrados motivaram o desenvolvimento dos processadores Super Escalares. Recentemente, após o lançamento no mercado de algumas dessas arquiteturas, o interesse de outros fabricantes e pesquisadores pelo problema de detecção e exploração do paralelismo de baixo nível (i.e., paralelismo a nível de instrução) foi estimulado.

Encontramos paralelismo de *alto nível* naqueles sistemas com dois ou mais processadores que executam simultaneamente diferentes trechos de um mesmo programa. O poder computacional provido por essa técnica de exploração de paralelismo é limitado basicamente pela rede de interconexão, pelo desempenho de cada um dos processadores do Sistema e pelo nível de dependência entre os fragmentos (do programa) que serão executados concorrentemente. O paralelismo de *baixo nível* que diz respeito as máquinas Super Escalares, explora por seu turno, a concorrência no interior de um único processador.

Nesse capítulo apresentamos as características das arquiteturas Super Escalares e mostramos que o nível de implementação (em *hardware* ou *software*) do algoritmo responsável pela detecção do paralelismo de programas, é usado para distinguir a classe que o processador pertence. Mostramos na seção seguinte, as técnicas de detecção de paralelismo e como as restrições impostas pelas dependências de dados e pela limitação de recursos do *hardware* interferem no processo da extração do paralelismo presente entre as instruções do código seqüencial. Na última seção

mostramos como a latência e o número de instruções despachadas por ciclo de processador, restringem o desempenho de arquiteturas Super Escalares.

2.2 Arquiteturas Super Escalares

Na literatura especializada, o termo *modo escalar* é empregado para distinguir a execução de uma única instrução, das instruções vetoriais que desencadeiam a ativação em paralelo de múltiplos elementos de processamento, todos executando a mesma operação. Por exemplo, instruções executadas em processadores convencionais como o Intel 80386, são do tipo *escalar*, enquanto que as executadas pelo ILLIAC IV [HORD82] são do tipo vetorial.

Arquiteturas Super Escalares são assim denominadas pois seguem um modelo alternativo de processamento no qual as instruções que são executadas concorrentemente não precisam possuir o mesmo código de operação, isto é, não são vetoriais.

Processadores vetoriais são arquiteturas SIMD (*Single-Instruction stream, Multiple-Data stream*) [FLYN66] adequadas para o processamento de aplicações científicas. Nessas aplicações, o programador costuma estruturar os dados em *arrays* cujos elementos são manipulados iterativamente, por estruturas de controle do tipo *Do Loop* [STON87]. Nesses processadores, graças aos inúmeros elementos de processamento, é possível a partir de uma única instrução vetorial, executar o equivalente a múltiplas iterações da estrutura de controle durante cada ciclo de máquina.

Assim como no processamento vetorial, máquinas Super Escalares possuem múltiplas unidades funcionais que podem ser ativadas em paralelo, resultando em um tempo de execução menor que o requerido pelo modelo seqüencial de processamento. Porém, enquanto no processamento vetorial as instruções concorrentes são do mesmo tipo, no modelo de processamento empregado pelas arquiteturas Super Escalares instruções de diferentes tipos podem estar sendo executadas simultaneamente. Por esse motivo, as arquiteturas Super Escalares além de atender a uma classe mais ampla de aplicações, conservam também as características de um processador de alto desempenho [JOUP89].

Embora executando múltiplas instruções em paralelo, arquiteturas Super Escalares não são máquinas do tipo MIMD (*Multiple-Instruction, Multiple Data stream*). Nessas arquiteturas, assume-se a existência de diversos fluxos de instruções que po-

dem ser oriundas (ou não) do mesmo programa de aplicação, e por esse motivo utilizam diversos contadores de programa (*program counter*). As arquiteturas Super Escalares possuem um único contador de programa e geralmente processam um único fluxo de instruções (procedente do mesmo programa).

Embora não seja visível ao programador do nível convencional, nas máquinas Super Escalares as instruções nem sempre são executadas na ordem em que foram especificadas originalmente. Existem duas fontes responsáveis por essa modificação na ordem de execução:

- a modificação pode ser levada a cabo durante a fase de otimização de código (realizada pelo compilador da linguagem de programação), isto é, o otimizador de código pode alterar a ordem de execução das instruções de modo a produzir código mais eficiente;
- a modificação pode ser provocada pelo mecanismo de *hardware* encarregado pela transferência das instruções para as diversas unidades funcionais do processador (i.e., pelo mecanismo responsável pelo despacho das instruções).

Em ambos os casos, é necessário que a reorganização do código preserve a *equivalência semântica* do programa, ou seja: o resultado final da execução do programa deve ser o mesmo para as duas seqüências de instruções.

Usando como critério a forma de detecção do paralelismo, podemos agrupar as máquinas Super Escalares em duas classes distintas: máquinas cujo algoritmo de detecção é diretamente implementado no *hardware*, e a classe das máquinas sem esse mecanismo.

No primeiro caso compete ao algoritmo implementado no *hardware*, a tarefa de determinar se dois ou mais comandos podem ser executados simultaneamente. Esse algoritmo é denominado *mecanismo de despacho de instruções*.

A existência de um algoritmo de despacho e de um sofisticado esquema de decodificação, viabilizam a detecção e extração de instruções que podem ser executadas simultaneamente, porém tornam as unidades de controle dessa classe de arquiteturas mais complexas. Independentemente da qualidade do código executado, o algoritmo do *hardware* tentará explorar mais efetivamente os recursos da máquina. Clássicos exemplos de máquinas com o algoritmo de detecção direta-

mente implementado no *hardware* incluem além das máquinas precursoras das arquiteturas Super Escalares (i.e., o CDC-6600 [THOR64] e o modelo IBM-360/91 [TOMA67]), os seguintes processadores: o IBM RS/6000 ([BAKO90] e [HEST90]); o Intel i960 ([HINT89]); o Motorola 88110 ([DIEF92] e [SMOT93]); a família *Power PC* do consórcio IBM/MOTOROLA/APPLE ([RYAN93], [THOM93], [CLYM94] e [STAM94]); o processador ALPHA da DEC ([DEC92] e [SITE93]), etc.

Para permitir a detecção de instruções que podem ser executadas em paralelo, o algoritmo do *hardware* precisa ter acesso a um bloco de instruções de máquina. Por essa razão, arquiteturas Super Escalares desse tipo geralmente incluem um *buffer* de instruções no interior do processador. Examinando as instruções desse *buffer*, o algoritmo de despacho tem condições de decidir se duas ou mais instruções podem ser executadas simultaneamente. Por exemplo, examinando o conteúdo do *buffer* de instruções, o algoritmo implementado no processador RS/6000 da IBM pode ativar a execução em paralelo de até cinco instruções [TABA91].

Nessa classe de processadores, técnicas de previsão de desvios assumem grande importância, já que a execução de instruções de desvio implica em queda de desempenho como descrito em [SMITH81], [JLEE84], [MCFA86] e [FERN92].

Uma descrição mais detalhada sobre essa classe de arquiteturas Super Escalares pode ser encontrada em [KELL75], [AUHT85], [ACOS86], [PLES88], [TABA91] [FERN92], [FERN92a] e [RAU93] entre outros. Contudo, nosso interesse maior nesse trabalho diz respeito a outra classe de arquitetura Super Escalar: a que não possui mecanismo de detecção implementado no *hardware*.

Máquinas Super Escalares sem algoritmo de detecção de concorrência, também são caracterizadas pela existência de um único fluxo de instruções e pela presença de múltiplas unidades funcionais que podem operar em paralelo. De modo distinto porém, sua unidade de controle é mais simples. Ela faz a busca do grupo de instruções que serão executadas durante cada ciclo do processador conforme especificado em tempo de compilação.

Denominamos ao conjunto de instruções buscado e executado em cada ciclo de processador, de “Instrução Multifuncional” (IMF) ou instrução longa. Portanto cada instrução longa (ou IMF) é formada pela coleção de instruções que terão sua execução iniciada no mesmo ciclo de máquina.

As arquiteturas Super Escalares sem mecanismo de despacho de instruções

podem ser diferenciadas, por meio da técnica empregada para ativar as unidades funcionais. Dentre elas temos:

- (1) instruções longas contendo campos distintos para controlar cada recurso da CPU;
- (2) instruções contendo somente alguns campos de controle;
- (3) código de operação especificando a execução paralela de dois ou mais comandos.

A primeira dessas três técnicas é empregada nos processadores do tipo LIW (*Large Instruction Word*) [HENN81] e VLIW (*Very Large Instruction Word*) [FISH83], [FISH84a], [ELLI86] e [COLW87]. A unidade de controle desses processadores é muito simples: para cada dispositivo funcional existe um campo na IMF indicando a função que deverá ser executada. A seguir ilustramos a organização das instruções longas de uma arquitetura VLIW.

Supondo que a máquina VLIW possui três ALUs (unidades realizando operações aritméticas e lógicas), e uma FPU (unidade realizando operações em ponto flutuante), então cada IMF deve incluir quatro campos distintos especificando a instrução que será realizada por cada ALU e pela FPU durante a execução da instrução longa. A Figura 2.1 mostra parte de uma IMF da máquina VLIW contendo os campos que controlam as três unidades aritméticas e a unidade de ponto flutuante.

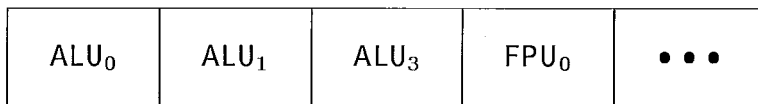


Figura 2.1: A Instrução Multifuncional (IMF) de uma Arquitetura VLIW

Na Figura 2.1 os três primeiros campos da IMF controlam as ALUs e o quarto campo a FPU da máquina VLIW em questão. Examinando um campo dessa instrução (Figura 2.2), notamos que nele estão presentes o código de operação e os operandos necessários para a execução da instrução especificada.

Na Figura 2.2, OP-*CODE* indica a instrução que deverá ser executada pela unidade funcional correspondente, *R_i* e *R_j* são os registradores fonte e *R_k* o destino da instrução.



Figura 2.2: Um Campo da Instrução Multifuncional de uma VLIW

Em tempo de execução, as instruções indicadas nos quatro campos da Figura 2.1 serão realizadas em paralelo. Códigos de operação do tipo NOP (*no operate*) deverão ser introduzidos se desejarmos que as unidades funcionais correspondentes não iniciem uma nova instrução quando a IMF for executada.

Conforme ilustrado, o programador exerce um controle direto sobre os recursos do *hardware* numa arquitetura VLIW. É através desse controle direto, técnica largamente empregada no nível de μ -programação [SALI76], que os programas de aplicação são codificados.

Na segunda técnica “poucas” instruções podem ser especificadas a partir de uma única IMF. Esse é o caso do processador MIPS desenvolvido em Stanford ([HENN81] e [KANE92]), onde até duas instruções podem ser empacotadas numa mesma IMF de 32 bits.

Na terceira técnica, o código de operação da instrução corrente pode indicar que a instrução adjacente deverá ser buscada e executada em paralelo. Por exemplo, no processador i860 da Intel o código de operação de uma instrução pode indicar o modo dual. Nesse modo de processamento, a unidade de controle do i860 inicia a execução em paralelo da instrução adjacente com a que especificou o modo dual de operação.

Arquiteturas sem algoritmo de despacho implementado no *hardware*, podem ser caracterizadas também pela ausência de mecanismos de sincronização (*interlocking mechanisms*). Ao invés disso, os projetistas transferem para os níveis hierárquicos superiores (i.e., para as camadas de *software*), a tarefa de sincronização. Problemas relacionados com as dependências de dados, com o tempo de latência das unidades funcionais, com os conflitos no uso dos recursos do *hardware* etc., usualmente são resolvidos antecipadamente, durante a fase de geração de código.

Conforme apontado por Fisher [FISH84], eliminando os circuitos lógicos responsáveis pela sincronização das atividades do processador, reduz-se a densidade

do circuito, liberando área para a inclusão de unidades funcionais e barramentos de dados adicionais.

Durante o escalonamento das instruções que serão executadas concorrentemente, o compilador deve levar em conta que um processador Super Escalar sem mecanismo de despacho, em geral incorpora as seguintes características ([COLW87] e [COWL88]):

- as instruções usam como operandos três registradores, exceto as instruções que referenciam a memória principal (*load / store*);
- banco global de registradores;
- ausência de mecanismos de sincronização;
- os recursos do *hardware* são expostos ao programador do nível convencional.

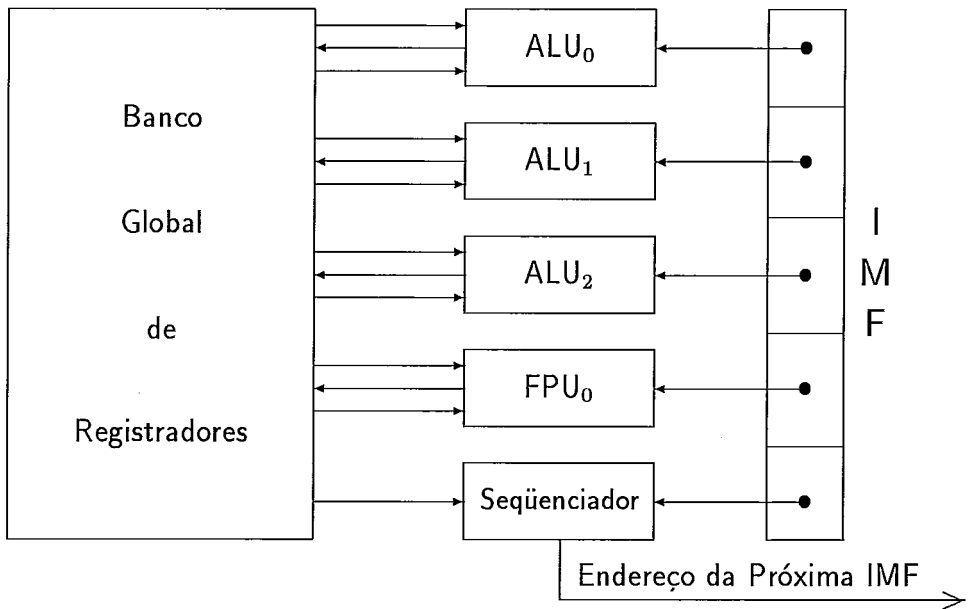


Figura 2.3: Componentes de uma Arquitetura VLIW

Podemos ilustrar os componentes da arquitetura VLIW cuja instrução multi-funcional mostramos na Figura 2.1, através do esquema da Figura 2.3.

A arquitetura VLIW da Figura 2.3 inclui: quatro unidades funcionais (três ALUs e uma FPU), um banco global de registrador com capacidade de múltiplos

acessos (*multi-port*) e a unidade de seqüenciamento. Cada instrução longa dessa máquina contém campos suficientes para especificar a operação que cada unidade funcional deve executar e quais os registradores usados como operandos. A IMF possui ainda um outro campo para especificar o tipo de seqüenciamento que será usado na geração do endereço da próxima IMF.

Existem algumas variações na implementação de máquinas sem algoritmo de *hardware* para a detecção de paralelismo. Essas variações afetam o esquema de geração de código, e por esse motivo também precisam ser consideradas pelo algoritmo de escalonamento de instruções.

Na implementação do modelo TRACE da Multiflow [COLW87] e do Cydra-5 da Cydrome [RAU89], ao invés de um único banco global de registradores, os projetistas incluíram vários bancos, cada um deles sendo diretamente acessado por um subconjunto de unidades funcionais.

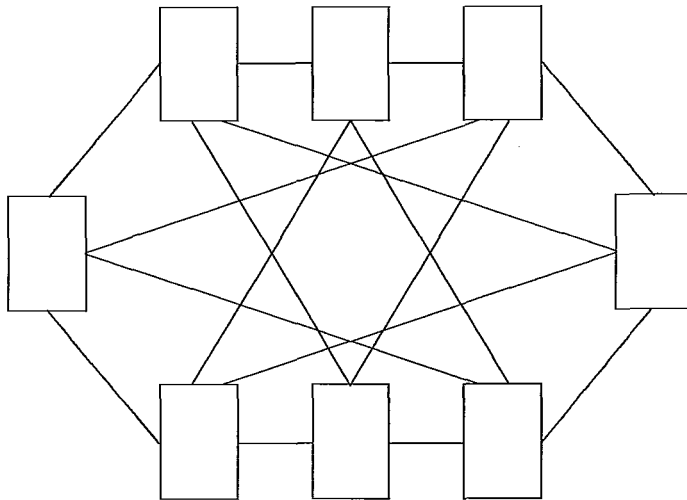


Figura 2.4: Esquema de Interconexão dos Módulos de uma Arquitetura VLIW

Se o registrador requerido para a execução de uma instrução não estiver no banco associado ao dispositivo funcional, o compilador gera código de máquina que irá transferir o conteúdo de um registrador remoto para a unidade funcional destino. Essa transferência é feita por uma unidade funcional diretamente conectada ao banco de registradores, e daí para a unidade destino através de um dos barramentos interconectando as duas unidades. A Figura 2.4 mostra o esquema de interconexão usado na transferência do conteúdo de um registrador armazenado num banco re-

moto.

Na Figura 2.4, os retângulos representam módulos de processamento de uma arquitetura VLIW e os segmentos de linhas retas representam barramentos interconectando os diversos módulos. Cada módulo inclui um banco de registradores e um grupo de unidades funcionais conectadas ao banco. Conforme podemos observar, existem barramentos conectando um módulo aos seus dois vizinhos mais próximos. Além dessas conexões, cada módulo possui dois outros barramentos de conexão com módulos não vizinhos. Através do esquema ilustrado, não mais do que dois módulos intermediários serão usados para transferir o conteúdo de um registrador num banco remoto para uma unidade funcional.

Uma outra variação na implementação de máquinas com escalonamento de instruções realizado pelo *software* de suporte, refere-se à capacidade funcional das múltiplas unidades. Uma possibilidade seria prover uma mistura de tipos de unidades funcionais, cada tipo especializado na execução de um subconjunto de instruções. O modelo TRACE da Multiflow é um exemplo de arquitetura Super Escalar com unidades funcionais heterogêneas. Por outro lado, podemos ter unidades funcionalmente homogêneas, ou seja, uma instrução pode ser executada por qualquer unidade. A máquina VLIW da IBM [EBCI88] e o modelo proposto por K. Anantha e F. Long [ANAN90] são exemplos de arquiteturas com unidades funcionais homogêneas.

Técnicas usadas para escalonar instruções para os respectivos dispositivos funcionais assumem um papel relevante no desempenho das arquiteturas Super Escalares. O principal objetivo dessas técnicas é gerar código capaz de manter as unidades funcionais bem utilizadas, aumentando desse modo, o desempenho do processador.

Essas técnicas, também denominadas de técnicas de detecção, baseiam-se na descoberta do paralelismo existente no código seqüencial a ser executado. Algoritmos implementados no *hardware* detectam concorrência em tempo de execução, enquanto que nos algoritmos implementados através de *software*, o paralelismo é detectado previamente, em tempo de compilação (ou manualmente pelo programador usando linguagem *Assembly*).

No escalonamento via *software*, o programa objeto é reorganizado pelo tradutor da linguagem de programação que seleciona as instruções que serão executadas concorrentemente. Conforme será visto na seção seguinte, durante o processo de reorganização, deve-se levar em consideração a disponibilidade dos recursos e a or-

dem de precedência de instruções dependentes.

2.3 Técnicas de Detecção de Paralelismo

As técnicas de escalonamento de instruções reorganizam o código objeto para manter as unidades funcionais das arquiteturas Super Escalares bem utilizadas, promovendo assim a redução no tempo de execução de programas do usuário. Independente da técnica empregada, o escalonamento deve considerar a interdependência das instruções (para preservar a equivalência semântica do código) e a limitação de recursos existente na arquitetura. O exemplo a seguir ilustra essa reorganização.

Tomemos como exemplo uma arquitetura VLIW hipotética, que possui dentre outras unidades funcionais, duas ALUs que realizam operações lógicas e aritméticas com inteiros.

Seja o seguinte trecho de programa cujas instruções devem ser executadas nas ALUs da arquitetura hipotética:

$$R1 := R2 + R3$$

$$R4 := R1 \times R5$$

$$R6 := R7 \text{ and } R8$$

Nesse caso, embora possam existir recursos disponíveis no momento da execução (já que temos duas ALUs), as duas primeiras instruções não podem ser ativadas concorrentemente. Se o algoritmo de escalonamento permitisse essa simultaneidade, a equivalência semântica do trecho de programa seria violada. Ou seja, a instrução $R4 := R1 \times R5$ só pode ser executada depois que a instrução $R1 := R2 + R3$ tiver modificado o conteúdo do registrador destino R1. Nesse caso a dependência de dados existente, obriga que a precedência do código seqüencial seja preservada.

Por outro lado, as instruções $R1 := R2 + R3$ e $R6 := R7 \text{ and } R8$ não apresentam dependência de dados e podem ser executadas simultaneamente, já que duas ALUs estão disponíveis na arquitetura.

Vamos examinar as questões da dependência de dados e da restrição de recursos mais detalhadamente, na Subseção 2.3.1.

2.3.1 Dependência de Dados e Restrições de Recursos

Dois tipos de dependências (ou interações) limitam o número de instruções que podem ser ativadas em paralelo: dependência de controle e dependência de dados. Dependência de controle (ou procedural) resulta do fluxo de controle do programa em execução [AUHT85].

Dependências procedurais são consequência da presença de instruções de desvio no código seqüencial. Qualquer instrução pode ser proceduralmente dependente de uma ou mais instruções de desvio, e somente de instruções de desvio. Um exemplo de dependência procedural pode ser visto no trecho de programa mostrado a seguir.

1. IF a > 0 GOTO 4.
2. b = c + d
3. e = f + g
4. x = y

A execução dos comandos 2. e 3. depende do comando 1. já que não é possível ativá-los até que a avaliação da condição ($a > 0$), seja conhecida.

A dependência de dados resulta do fluxo de dados durante a execução de um programa: o conteúdo das variáveis é alterado pelos comandos de atribuição.

Formalmente, a dependência de dados pode ser descrita do seguinte modo:

Sejam F_k o conjunto dos operandos fonte, e D_k o conjunto dos operandos destino da instrução k . Existe dependência de dados entre duas instruções i e j , onde i ocorre antes de j no código seqüencial **se e somente se**, pelo menos uma das condições a seguir for verdadeira:

1. $D_i \cap F_j \neq \emptyset$
2. $F_i \cap D_j \neq \emptyset$
3. $D_i \cap D_j \neq \emptyset$

Caso contrário, dizemos que j não depende de i .

Em outras palavras, se no código seqüencial, a instrução i aparece antes da instrução j , e se pelo menos uma das condições anteriores for verdadeira, então para

que a equivalência do código seja preservada, a instrução i tem que ser executada antes da instrução j .

Dizemos que a dependência de dados causada pela Condição 1 é uma *dependência verdadeira*, que a causada pela Condição 2 é uma *anti-dependência*, e que a causada pela Condição 3 é uma *dependência de saída*.

Para ilustrar cada uma das condições que provocam dependência de dados, mostramos na Tabela 2.1, três trechos de programas. Nos três trechos, o uso da variável A acarreta que os dois comandos de cada trecho, sejam dependentes.

Condição 1	Condição 2	Condição 3
\dots $i. \quad A := B + 1$ $j. \quad C := A \times 2$ \dots	\dots $i. \quad C := A \times 2$ $j. \quad A := B + 1$ \dots	\dots $i. \quad A := B + 1$ $j. \quad A := C \times 2$ \dots

Tabela 2.1: Exemplos de Dependência de Dados

No primeiro trecho o conjunto dos operandos destino da instrução i ($A := B + 1$), é $\{A\}$, ao passo que na instrução seguinte, j ($C := A \times 2$), o conjunto de operandos fonte é $\{A\}$, portanto a Condição 1 é verdadeira, isto é, a interseção entre os conjuntos dos operandos de destino de i , e o dos operandos fonte de j não é vazia. Temos um exemplo de dependência verdadeira ou RAW (*Read After Write*).

No segundo trecho da Tabela 2.1, o conjunto dos operandos fonte da instrução i ($C := A \times 2$), é $\{A\}$, ao passo que na instrução seguinte, j ($A := B + 1$), o conjunto de operandos destino é $\{A\}$, portanto a Condição 2 é verdadeira, isto é, a interseção entre os conjuntos dos operandos de fonte de i , e o dos operandos destino de j não é vazia. Temos um exemplo de anti-dependência ou WAR (*Write After Read*).

Finalmente no terceiro trecho de programa da Tabela 2.1, o conjunto dos operandos destino da instrução i ($A := B + 1$), é $\{A\}$, ao passo que na instrução seguinte, j ($A := C + 2$), o conjunto de operandos destino é $\{A\}$, portanto a Condição 3 é verdadeira, isto é, a interseção entre os conjuntos dos operandos de destino de i , e o dos operandos destino de j não é vazia. Temos um exemplo de dependência de saída ou WAW (*Write After Write*).

A partir desses exemplos, podemos verificar que as dependências de dados

e procedurais estabelecem uma *relação de precedência* entre os comandos de um programa, isto é, o início de um comando pode estar condicionado ao término de um outro que o precede. Desse modo, podemos ver que as dependências atuam como barreiras, forçando o seqüenciamento dos comandos de um programa.

Concluimos então que as dependências são parcialmente responsáveis pela queda no desempenho de processadores com paralelismo de granularidade fina como é o caso das arquiteturas Super Escalares.

Diversas técnicas de otimização para eliminar dependências têm sido desenvolvidas. Essas técnicas conseguem eliminar os efeitos da anti-dependência e da dependência de saída [MOON92], porém o efeito provocado pela dependência verdadeira, não pode ser eliminado.

Para facilitar a detecção e tratamento das dependências dos comando de um programa, costuma-se empregar uma estrutura de dados, denominada grafo de dependência de dados (*Data Dependence Graph*, ou simplesmente DDG). Os comandos do programa são representados por um nó do grafo, e as arestas (que são orientadas) correspondem às relações de precedência dos comandos. Por exemplo, o DDG de cada um dos três trechos de programas apresentado na Tabela 2.1 é assim representado:



Para construir o DDG de um programa precisamos conhecer os conjuntos de operandos fonte e destino das instruções, ordem de execução dos comandos e as relações de dependência determinadas pelas Condições 1, 2 e 3.

Usamos a relação precede (**prcd**) para denotar a ordem de execução das instruções de um programa. Dessa forma, a relação i **prcd** j (i precede j), indica que o comando i deve ser executado antes do comando j . Basicamente, a relação **prcd** estabelece a ordem de execução dos comandos de um programa. Nos três trechos de

programas mostrados da Tabela 2.1, dizemos que i **prcd** j .

Antes de examinarmos como é feita a construção do grafo de dependência de dados, vamos apresentar alguns conceitos necessários para a manipulação das técnicas de escalonamento de instruções.

Sabemos que o problema de escalonamento de instruções para um processador VLIW é análogo ao problema da compactação de μ -código para μ -máquinas com μ -instruções horizontais ([FISH81] e [FERN92]). Por conveniência, vamos usar o termo “compactação de instruções,” no lugar de compactação de μ -código.

Add R1,R2,R3	Mul R4,R5,R6	Sub R7,R8,R9	Fadd R0,R11,R12	• • •
--------------	--------------	--------------	-----------------	-------

Figura 2.5: Instrução Longa de uma Arquitetura VLIW

Como vimos na Seção 2.2, uma instrução multifuncional (ou instrução longa), é formada por um conjunto de instruções que podem ativar concorrentemente todas as unidades funcionais do processador. Na Figura 2.5 vemos o trecho de uma instrução longa capaz de ativar as três ALUs e a FPU de uma arquitetura VLIW hipotética. Nesse caso durante o processo de compactação foram encontradas instruções não dependentes capazes de ativar os quatro dispositivos de processamento da arquitetura. Em situações em que não é possível ativar todos os dispositivos, ou quando é necessário reservar uma unidade funcional já ativada por uma instrução que iniciou num ciclo anterior e que ainda não terminou, então a instrução NOP deve ser inserida na IMF, no campo correspondente ao dispositivo.

Conhecendo então a diferença entre uma instrução e uma instrução multifuncional, podemos reescrever a definição para o problema da compactação de μ -código [LAND80], de modo adequá-lo ao problema do escalonamento de instruções.

O problema da **Compactação de Código Seqüencial** pode ser assim definido:

- Vamos supor que para uma determinada máquina Super Escalar, tenhamos um programa descrito como uma seqüência de instruções. Escalonar essas instruções consiste em alocá-las nas instruções multifuncionais da máquina Super Escalar de forma que o tempo de execução do programa

seja minimizado. O processo de alocação (ou compactação) das instruções nas IMFs deve garantir que a seqüência resultante de IMFs seja semanticamente equivalente à seqüência original de instruções.

- Definição: Uma seqüência de instruções multifuncionais $S1$ é semanticamente equivalente a seqüência de instruções $S2$, se o mesmo resultado k é produzido, quando a seqüência $S1$ ou a seqüência $S2$ é executada.

Do mesmo modo que na compactação de μ -código as técnicas de escalonamento de instruções podem ser classificadas como locais ou globais. Técnicas de compactação local atuam em trechos contínuos de código, ou seja, em blocos básicos. Por esse motivo, torna-se necessário delimitar as fronteiras dos blocos básicos.

- Definição: Denominamos de Bloco Básico (ou Trecho Contínuo de Código), uma coleção ordenada de instruções sem pontos de entrada (*entry points*), exceto no início do trecho e sem comandos de desvio, exceto se for a última instrução do trecho.

A Figura 2.6 apresenta os blocos básicos de um trecho de programa.

```

1.  A := B × C;
2.  D := 63;
3.  If A ≥ D then Goto 7
-----
4.  E := A or 15;
5.  F := D - E;
6.  Goto 9;
-----
7.  E := A and 31;
8.  F := E × 2;
-----
9.  B := B + 1;
    • • •
-----

```

Figura 2.6: Blocos Básicos de um Programa

As nove instruções do trecho de programa apresentado na Figura 2.6 formam quatro blocos básicos (delimitados pelas linhas horizontais). Ao delimitar essas fronteiras, estamos assumindo que o ponto de entrada de cada um dos blocos corresponde ao primeiro comando, isto é, instruções 1, 4, 7 e 9. Em outras palavras, o

fluxo de controle não pode ser transferido para nenhuma outra instrução do bloco. Se tal ocorresse, estaríamos violando a definição de Bloco Básico.

Usando a lista de instruções do bloco básico em conjunto com as relações de dependências, podemos construir o respectivo grafo de dependência de dados. O algoritmo de construção do grafo examina uma instrução de cada vez, iniciando com a primeira instrução do bloco básico, e percorre a lista de instruções seqüencialmente até o final do trecho. Para cada instrução examinada, um novo vértice é inserido no grafo. Em seguida, percorremos o grafo para determinar os vértices que devem ser conectados ao novo nó. Nesse caso, dizemos que estamos inserindo a instrução corrente no grafo corrente. O procedimento completo para construção do DDG está descrito em [FERN92].

Quando da geração do DDG, não levamos em consideração as características da arquitetura onde o programa será executado. Por essa razão, apesar do DDG mostrar que duas instruções, I_j e I_k são independentes, nem sempre elas podem compartilhar uma mesma instrução longa. Esse impedimento decorre de limitações impostas pelo *hardware* disponível. Por exemplo, apesar de não dependentes, duas instruções que utilizam a única ALU do processador não podem ser empacotadas na mesma IMF. Essa situação, denominada de conflito, deve ser considerada durante o processo de compactação.

Com o objetivo de determinar a ocorrência de conflitos na utilização dos recursos da máquina, precisamos gerar uma descrição dos componentes do *hardware* que são utilizados pelas diversas instruções. Tal descrição deve incluir detalhes relacionados com a temporização da instrução, pois no caso de arquiteturas polifásicas [SALI76] é possível alocar um mesmo recurso (do tipo não compartilhável) por duas ou mais OPs desde que ele seja utilizado em sub-ciclos distintos. Esse é um caso especial de dependência, denominada dependência fraca, e que pode ser assim definida:

Definição: Duas instruções são fracamente dependentes se elas forem dependentes e se os recursos responsáveis pela dependência forem liberados pela OP precedente antes do início da instrução subsequente.

A especificação dos recursos do processador e da sua temporização pode ser feita por intermédio da descrição de suas operações sob a forma de 6-*uplas* descritas em [AMOR88] e [FERN92], do tipo:

$$(Id, E, S, U, T, C),$$

onde os componentes são:

- Id: identificador da instrução;
- E: conjunto dos registradores usados como operandos de entrada pela instrução;
- S: conjunto dos registradores usados como saída pela instrução;
- U: conjunto das unidades funcionais utilizadas durante a execução da instrução;
- T: conjunto dos sub-ciclos requeridos para a execução da instrução;
- C: conjunto dos campos da instrução utilizados na sua representação, incluindo seus dados imediatos.

Para compactar uma instrução na IMF, precisamos fazer a interseção da sua $6 - \text{upla}$ com as $6 - \text{uplas}$ das instruções já alocadas na instrução longa. Se a interseção for vazia, então não existe conflito, e a instrução pode ser alocada nessa instrução multifuncional. Caso contrário, é necessário verificar se as interseções de todos os sub-ciclos são vazias, ou seja, se o recurso pode ser usado por mais de uma instrução em diferentes sub-ciclos da mesma IMF.

2.3.2 A Compactação

As máquinas Super Escalares, sem mecanismo de detecção de paralelismo implementado diretamente pelo *hardware*, resultam do desenvolvimento das técnicas da compactação de μ -código. Tal como ocorre com as μ -máquinas com μ -instruções horizontais, a grande barreira responsável pela redução no nível de paralelismo que pode ser extraído de uma arquitetura Super Escalar, consiste na dificuldade de programá-la.

Tendo em vista que o escalonamento das instruções de um processador Super Escalar recai no problema da compactação de μ -programas, podemos então utilizar as diversas técnicas de compactação de μ -código.

Para que o escalonamento das instruções de programa seja realizado, os seguintes passos precisam ser levados a cabo:

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

O grafo de dependências e a descrição dos recursos são estruturas consultadas pelo algoritmo de escalonamento: antes de empacotar uma instrução na IMF, o algoritmo verifica se a movimentação pode ser feita, ou seja, é verificado se a semântica do programa será preservada e se os recursos de *hardware* que serão utilizados durante a execução da instrução estarão disponíveis durante o respectivo ciclo de máquina.

O algoritmo de compactação recebe como entrada uma seqüência ordenada de instruções, e produz como resultado o programa compactado. Cada elemento da seqüência de entrada é constituído por uma única instrução, conforme ilustrado no exemplo a seguir:

1. $A := B \times C;$
2. $D := E + F;$
3. $G := G + 1;$
4. If $A \geq G$ then goto 11;

• • •

A seqüência do exemplo é constituída por quatro comandos. Ao receber como entrada esse trecho, o algoritmo de compactação poderia produzir como resultado a seguinte seqüência:

- (i) $A := B \times C; \quad D := E + F; \quad G := G + 1;$
- (ii) If $A \geq G$ then goto $x;$

• • •

Podemos constatar que cada componente da seqüência resultante é formada por uma combinação das instruções especificadas na entrada. Podemos verificar também que as três primeiras instruções da seqüência original serão executadas simultaneamente quando da ativação da IMF i . Tendo em vista que a rotulação das IMFs foi modificada pelo algoritmo, então o endereço alvo do desvio (goto 11

originalmente) foi fixado para um novo valor x .

A seqüência original pode ser produzida manual ou automaticamente (por um compilador). A segunda seqüência é produzida pelo algoritmo de compactação, e corresponde ao programa objeto de um processador Super Escalar.

Como dissemos anteriormente, dois tipos de estratégias básicas podem ser usadas durante o processo de compactação (ou escalonamento): compactação Local e Global.

Na Compactação Local, o alcance das atividades do algoritmo restringe-se aos comandos do bloco básico. Somente após o tratamento de um bloco básico, é que o algoritmo de compactação local examina um outro bloco.

As técnicas de Compactação Global examinam o programa integralmente. Elas não se restringem as fronteiras impostas pelos blocos básicos, e por esse motivo é possível migrar instruções de um bloco básico para um outro. Em outras palavras, algoritmos de compactação global permitem empacotar um número maior de instruções nas instruções multifuncionais, e por esse motivo, uma taxa mais significativa de concorrência pode ser extraída.

A Compactação Local

Na Compactação Local, após determinar os blocos básicos do programa, o algoritmo de escalonamento associa uma instrução longa (inicialmente vazia) para cada um dos comando do bloco básico, e em seguida inclui o comando na sua respectiva instrução longa. Durante o processo de escalonamento, o algoritmo de compactação local examina cada comando do bloco básico, verificando se é possível movimentá-lo para a IMF precedente. Se o comando for movimentado então ele será executado em paralelo com os outros comandos do bloco básico já alocados naquela IMF. Conflitos no uso de recursos (operandos ou dispositivos funcionais) podem impedir que dois ou mais comandos compartilhem a mesma instrução longa. Nesse caso, o algoritmo de compactação precisa alocá-los em IMFs distintas. A ordem em que os comandos do bloco básico são examinados varia de acordo com o algoritmo de escalonamento empregado, e geralmente afeta a qualidade do código compactado [LAND80].

Diversos algoritmos de compactação local foram especificados e implementados. Landskov [LAND80] classifica-os em quatro grupos:

- (1) FCFS – as instruções são incluídas na lista de instruções conforme sua

ordem original no bloco básico (esquema *first come first served*);

- (2) Caminho Crítico – a ordem de inclusão da instrução leva em consideração sua posição no DDG. A instrução com o caminho mais longo possui maior prioridade no atendimento;
- (3) Separação e Avaliação (*Branch and Bound*) – classe geral de algoritmos de escalonamento baseados nas técnicas de busca em árvores. Em compactação, esses algoritmos fazem a alocação das instruções durante a construção da árvore. Os nós da árvore são as instruções longas, e um caminho do nó raiz até uma folha corresponde a uma lista de instruções longas. Sempre que existir mais de uma instrução longa que pode ser colocada no nó sucessor, a árvore bifurca-se. No final da construção da árvore, teremos todas as possíveis combinações de instruções longas, podendo-se escolher a combinação ótima;
- (4) Lista de Escalonamento – essa classe é um caso particular do grupo anterior com heurística podando ramos da árvore. Ao invés de gerar todas as possíveis combinações de instruções longas, a heurística empregada irá selecionar algumas combinações. Conseqüentemente, a obtenção da combinação ótima não é garantida. Contudo, o tempo de compactação pode ser substancialmente reduzido. O critério proposto por Wood [WOOD78] consiste em associar pesos a cada uma das instruções. Esses pesos são atribuídos de acordo com o número de descendentes da instrução no DDG. Os pesos são usados para selecionar a instrução que será incluída na instrução longa.

As instruções de desvio, que delimitam blocos básicos, ocorrem em programas de aplicação com uma freqüência considerável, representando de 15 a 30% das instruções executadas pelos processadores ([SHUS77], [GROS82], [MCFA86] e [DAVI90]). Esse elevado percentual de ocorrência reduz os blocos básicos a poucas instruções, limitando a taxa de concorrência que pode ser extraída do código seqüencial, quando técnicas de compactação local são empregadas.

A Compactação Global

Na Compactação Global o algoritmo movimenta instruções além das fronteiras demarcadas pelos blocos básicos. Por esse motivo, o algoritmo precisa de uma estrutura adicional, denominada grafo do fluxo de controle do programa (*control-*

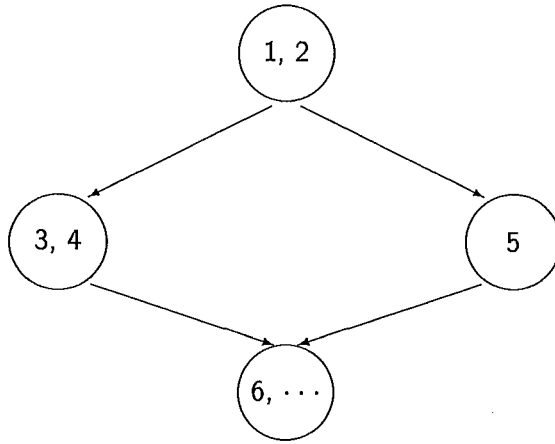


Figura 2.7: Grafo do Fluxo de Controle de um Programa

flow graph). Examinando o grafo do fluxo de um programa e obedecendo algumas regras básicas de movimentação, o algoritmo realiza a compactação global.

Os vértices do grafo do fluxo de controle correspondem aos blocos básicos do programa, e as arestas são assim geradas: o grafo contém uma aresta do bloco B_i para o bloco B_j se o fluxo de controle pode ser transferido de B_i para B_j . Esse grafo deve ser acíclico, ou seja, precisamos eliminar os laços (*loops*). A Figura 2.7 mostra o grafo do fluxo de controle do trecho de programa apresentado na Figura 2.8. Na Figura 2.8 os blocos básicos estão delimitados pelas linhas horizontais.

```

1.  sum := sum + 1;
2.  if (i and 1) <> then goto 5;
-----
3.  odd := odd + 1;
4.  goto 6;
-----
5.  even := even + 1;
-----
6.  i := i + 1;
   • • •
-----
  
```

Figura 2.8: Blocos Básicos de um Trecho de Programa

O grafo do fluxo de controle ilustrado na Figura 2.7 inclui quatro vértices e quatro arestas. A partir do primeiro bloco (o que contém os comandos 1 e 2) o controle pode ser transferido para o segundo e o terceiro bloco. Por esse motivo introduzimos arestas interconectando os vértices $\{1, 2\}$ e $\{1, 3\}$. Analogamente,

foram introduzidas arestas interconectando os vértices $\{2, 4\}$ e $\{3, 4\}$.

Diversos algoritmos de compactação global foram especificados e implementados. Entre eles o algoritmo apresentado por Tokoro [TOKO81], a técnica *trace scheduling*, desenvolvida em 1981 por Fisher [FISH81] e o *Percolation Scheduling* proposto por Aiken e Nicolau [NICO85]. O *trace scheduling*, destinava-se originalmente à compactação global de μ -código horizontal. Posteriormente, o algoritmo foi adaptado e incorporado no compilador Buldog [ELLI86], desenvolvido na Universidade de Yale, com a finalidade de gerar código para a arquitetura VLIW ELI-512 [FISH83]. Podemos também mencionar a técnica de compactação empregada no reorganizador de código do processador MIPS (*Microprocessor without Interlocked Pipeline Stages*), desenvolvido na Universidade de Stanford [HENN83]. Resumidamente podemos dizer que:

- (1) O algoritmo de compactação global apresentado por M. Tokoro movimenta instruções entre blocos adjacentes, segundo quatro regras básicas por ele propostas [TOKO81];
- (2) *Trace scheduling* é uma técnica de compactação global que faz o escalonamento das instruções ao longo do fluxo de controle do programa como um todo. Isso significa que a técnica explora o paralelismo em trechos (*traces*) de execução do programa. A escolha dos trechos do programa compactados prioritariamente, baseia-se na probabilidade com que estes são executados. A probabilidade de execução de cada trecho do programa, desempenha um importante papel na qualidade do código gerado, e pode ser fornecida pelo programador, ou determinada automaticamente. A movimentação de instruções ao longo dos *traces*, obriga a inclusão de código de reparo [ELLI86] (isto é, código para anular os efeitos da movimentação de instruções para outros blocos básicos), na última fase do processo de compactação;
- (3) *Percolation Scheduling* movimenta os comandos do programa de aplicação além das fronteiras que delimitam os blocos básicos. Esse processo de compactação é levado a cabo pelas quatro primitivas que movimentam comandos entre vértices adjacentes do grafo que representa o programa de aplicação. As primitivas de movimentação do algoritmo preservam a equivalência semântica em todos os caminhos do grafo que são afetados pelas transformações. São elas: *delete* (elimina um vértice do grafo),

move-op (movimenta comando para um vértice adjacente), *move-cj* (movimenta um comando de desvio condicional), e *unification* (unifica diversos comandos idênticos). Quando se faz necessário, código de reparo é acrescentado ao código original;

- (4) O Reorganizador de Código do processador MIPS compacta até duas instruções numa instrução multifuncional. As instruções de desvio nesse processador são do tipo *delayed branch* (FERN92), com retardo de um ciclo. Por essa razão o Reorganizador tenta empacotar uma outra instrução com a instrução de desvio.

Todas as técnicas citadas, empregadas na compactação global estão descritas detalhadamente em [FERN82].

A aplicação de algoritmos de compactação global permite empacotar um número maior de instruções nas IMFs. Conseqüentemente, uma taxa maior de concorrência pode ser extraída do código original. Devemos contudo ter em mente que algumas desses algoritmos provocam explosão no tamanho do código objeto. A razão dessa explosão deve-se ao código de reparo necessário para que a equivalência semântica seja preservada.

2.4 Latências e Número de Instruções Despachadas por Ciclo

Existem ainda outros fatores que limitam o desempenho das arquiteturas Super Escalares que merecem ser analisados. Nessa seção, vamos examinar o efeito provocado por dois desses fatores, a latência e o número de instruções despachadas por ciclo de processador, no desempenho dessas arquiteturas.

Denomina-se de tempo de latência de uma instrução (ou simplesmente latência), o intervalo de tempo requerido para a sua execução. Esse intervalo de tempo varia de acordo com a complexidade da instrução, e em alguns casos com os valores dos operandos de entrada. Por exemplo, a latência de uma instrução de multiplicação é maior do que o tempo de latência de uma operação lógica. Além disso, dependendo da forma como a multiplicação foi implementada, o tempo de latência da instrução pode variar em função do número de bits do multiplicador que forem iguais a zero, isto é, o resultado de 35×4 pode ser obtido mais rapidamente do que o resultado

de 35×3 .

Com o objetivo de simplificar o projeto de processadores (principalmente aqueles do tipo RISC), o tempo de latência da instrução mais complexa é usado para determinar o ciclo de máquina. Apesar de facilitar a especificação do processador, essa estratégia de projeto pode aumentar o tempo de processamento. A unidade de controle permanecerá ociosa após a execução de instruções com latências menores, aguardando pelo transcurso do tempo de ciclo do processador.

No projeto de uma arquitetura Super Escalar, a espera pelo transcurso do maior tempo de latência deve ser evitada, caso contrário, teríamos uma queda considerável na utilização dos recursos do *hardware*. Uma redução no nível de ociosidade das unidades funcionais provoca um aumento no desempenho do processador. Em contrapartida, esse aumento na eficiência do processador é acompanhado pela complexidade do algoritmo responsável pelo tratamento de latências diferenciadas, que é significativamente maior.

O exemplo que mostramos a seguir ilustra o efeito da latência no desempenho de processadores.

Vamos considerar um processador hipotético, que possui três unidades funcionais: uma de adição, uma de multiplicação e uma que executa as operações lógicas. Cada uma das unidades funcionais é caracterizadas pelo seu tempo de latência, ou seja:

- adição: 3 unidades de tempo;
- multiplicação: 5 unidades de tempo; e
- operações lógicas: 2 unidades de tempo.

As três unidades funcionais presentes usam como operandos três registradores do banco de registradores da arquitetura, e cada instrução que executam, possui o seguinte formato:

$$R_d := R_{f1} \text{ Op } R_{f2};$$

onde R_d , R_{f1} e R_{f2} são respectivamente operandos destino e fontes, e **Op** é código da operação.

O projetista da arquitetura descrita, com o objetivo de simplificar o projeto, decidiu definir que a latência de qualquer instrução executada pelo processador é

igual ao maior tempo de latência existente. O ciclo desse processador é portanto de cinco unidades de tempo.

Na Figura 2.9 podemos verificar o efeito causado pela execução seqüencial combinado com o alongamento das latências para cinco unidades de tempo. O diagrama de temporização mostrado na Figura 2.9 diz respeito a execução do trecho de programa da Figura 2.10.

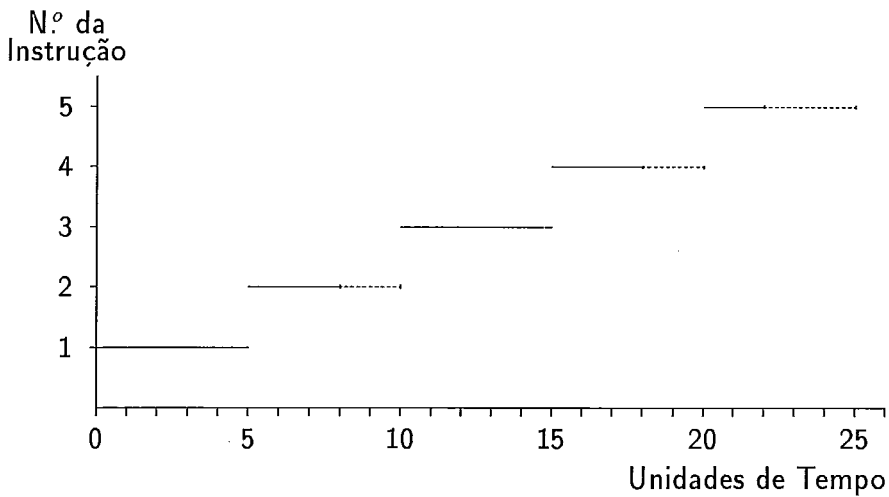


Figura 2.9: Temporização na Execução Seqüencial com Latências Iguais

- | | |
|----|-------------------------------|
| 1. | $R_0 := R_1 \times R_2$ |
| 2. | $R_3 := R_4 + R_5$ |
| 3. | $R_6 := R_7 \times R_8$ |
| 4. | $R_9 := R_2 + R_6$ |
| 5. | $R_9 := R_1 \text{ and } R_4$ |

Figura 2.10: Instruções para a Arquitetura Hipotética

Na Figura 2.9, cada segmento de reta contínuo representa o intervalo de tempo durante o qual a correspondente unidade funcional permaneceu ocupada, executando uma das cinco instruções. O intervalo de tempo em que o processador ficou ocioso é representado na figura por segmentos de reta tracejados, e o seu valor é igual a $5 - T_{lu}$, onde T_{lu} é o tempo de latência da unidade funcional que realizou a operação especificada pela instrução. Por exemplo, para a instrução 2, esse tempo é igual a $5 - 3$ unidades. Somando os intervalos de tempo durante os quais o processador ficou ocioso, verifica-se que dentre as 25 unidades de tempo de processamento do trecho

de programa, sete unidades foram desperdiçadas por causa da decisão do projetista em fixar o ciclo de máquina como sendo igual ao tempo de latência da unidade de multiplicação.

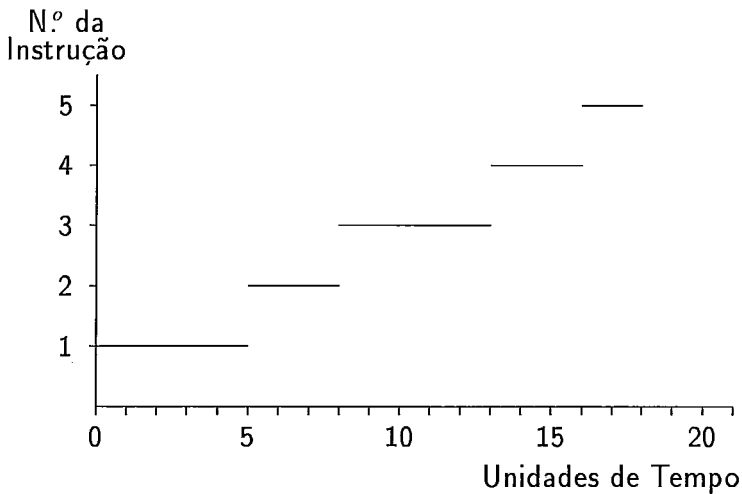


Figura 2.11: Temporização na Execução Sequencial com Latências Diferentes

Na Figura 2.11 mostramos como o tempo de execução das instruções da Figura 2.10 cai quando as latências das unidades funcionais do processador não são distendidas para o tempo de latência da multiplicação, isto é, para cinco unidades de tempo. Na Figura 2.11 vemos que o tempo total de execução das cinco instruções é reduzido de 25 unidades (Figura 2.9), para 18 unidades. Essa redução deve-se ao despacho da instrução 3 no tempo $t = 8$, ao invés de aguardar o transcurso de duas unidades de tempo adicionais, durante as quais a unidade responsável pela adição que executa a instrução 2, permanece ociosa. Situação idêntica ocorre quando do despacho da instrução 5, isto é, é despachada no tempo $t = 16$, e não no tempo $t = 20$. Finalmente a instrução 5, é executada em apenas duas unidades de tempo ao invés cinco unidades de tempo, reduzindo o tempo de execução de um total de 7 unidades de tempo.

Podemos agora supor, que a nossa arquitetura hipotética é uma VLIW e que portanto possibilita a execução concorrente de instruções. Vamos então examinar o efeito da execução paralela, considerando que a cada ciclo do processador uma nova IMF é buscada e a execução das suas instruções é iniciada. Nessa situação, temos como resultado o diagrama de temporização mostrado na Figura 2.12, quando da execução do trecho de programa da Figura 2.10.

A disponibilidade de unidades funcionais e a independência das instruções 1,

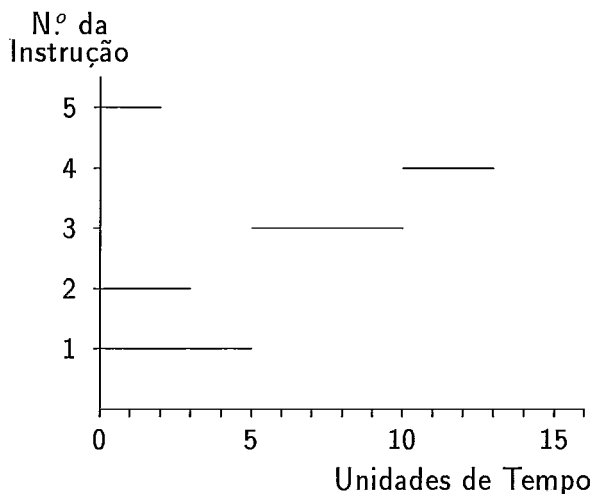


Figura 2.12: Ativação de Múltiplas Instruções por Ciclo com Latências Diferentes

2 e 5 permitiram a ativação simultânea dessas três instruções no tempo $t = 0$. A ativação em paralelo das instruções 1 e 5 somente foi possível em virtude de termos assumido que o registrador R_1 foi transferido para a unidade de multiplicação antes de ser alterado pela instrução 5. Tendo em vista a indisponibilidade da unidade de multiplicação (permaneceu ocupada, executando a instrução 1 durante as cinco primeiras unidades de tempo de processamento), então podemos ver no diagrama de temporização da Figura 2.12 que a instrução 3 somente foi ativada no tempo $t = 5$. A dependência verdadeira entre as instruções 3 e 4, impediu o despacho da instrução 4 no tempo $t = 3$, retardando sua ativação para o tempo $t = 10$, que é quando o novo conteúdo do registrador R_6 tornou-se disponível.

Examinando o diagrama de temporização da Figura 2.12, podemos constatar que ocorreu redução no tempo de processamento do trecho de programa em relação ao esquema apresentado na Figura 2.11. Ao invés dos 18 ciclos de máquina, na VLIW somente 13 ciclos de processador foram requeridos. Essa redução decorre da execução em paralelo das instruções 1, 2 e 5.

É importante ressaltar que embora considerável a redução no tempo de processamento, o pequeno tamanho do trecho de programa apresentado e as dependências de dados das instruções são responsáveis pela baixa utilização dos recursos do hardware. Assim, somente ao longo da execução das três primeiras instruções é que as unidades funcionais permaneceram ativadas simultaneamente. Mais precisamente, somente para valores do tempo $t \in [0, 2]$, é que as três unidades funcionais permaneceram ocupadas em paralelo. Podemos ainda verificar na Figura 2.12, que

apenas para $t \in [2, 3]$ é que exatamente duas unidades estiveram operando concorrentemente (no caso, a unidade de adição e a de multiplicação). Finalmente, para $t \in [3, 13]$ não mais do que uma unidade funcional permaneceu em execução.

Nas arquiteturas VLIW, o número de instruções despachadas e executadas simultaneamente varia conforme o modelo e o fabricante. O processador ELI por exemplo (vide [FISH83] e [FISH84a]), desenvolvido na Universidade de Yale, inclui oito controladores de memória e oito unidades funcionais, e por esse motivo, até 16 instruções podem ser despachadas e executadas concorrentemente. Já nos modelos Trace 7, Trace 14 e Trace 28 da Multiflow Trace, as 7, 14 e 28 unidades funcionais podem ser ativadas simultaneamente, além de 1, 2 e 4 instruções de desvio.

Finalmente é preciso lembrar que o desempenho das arquiteturas Super Escalares é limitado pela quantidade de paralelismo de baixo nível presente nos programas de aplicação ([WALL91], [LAM92]).

No Capítulo 3 apresentamos nosso modelo de arquitetura Super Escalar VLIW, que possibilita a execução condicional de instruções compactadas em uma mesma instrução multifuncional.

Capítulo 3

O Modelo

3.1 Introdução

Nesse capítulo examinamos aspectos relativos ao modelo de processador Super Escalar que permite execução condicional de instruções empacotadas em instruções longas do programa executável. Inicialmente descrevemos o código intermediário produzido pelo compilador a partir do código fonte, a ser paralelizado durante o processo de compactação. Apresentamos na seção seguinte o conceito da execução condicional. Na Seção 3.4 mostramos o modelo da VLIW proposto, isto é, seus componentes, a interligação existente entre os mesmos e a latência de cada uma das suas unidades funcionais. Destinamos as Seções 3.5 e 3.6 para detalhar respectivamente, os elementos da arquitetura e a interconexão entre eles. Finalmente na última seção, mostramos como simulamos o modelo apresentado.

3.2 O Código Intermediário

Nessa Seção descrevemos o código intermediário seqüencial que utilizamos para gerar programas objeto, executáveis no nosso modelo de processador Super Escalar.

A linguagem de alto nível reconhecida pelo compilador responsável pela geração da forma intermediária, é um subconjunto da linguagem Pascal. A implementação do compilador, a definição do código intermediário, e a escolha do subconjunto da linguagem Pascal foram atividades levadas a cabo por Leonardo Cardoso Monteiro num trabalho anterior [MONT93].

O subconjunto da linguagem Pascal reconhecida pelo compilador inclui os tipos *Boolean*, *Char*, *Integer*, *Real* e *Arrays* de uma dimensão. As estruturas para controle de fluxo *For* e *Case* e a definição de funções e procedimentos recursivos não foram incluídas na linguagem.

O compilador traduz o código fonte para um código intermediário composto por 28 instruções distintas. A forma geral dessas instruções é:

CÓDIGO DE OPERAÇÃO	[DESTINO]	OPERANDO ₁	...	OPERANDO _n
--------------------	-----------	-----------------------	-----	-----------------------

As instruções da forma intermediária enquadram-se nas seguintes categorias:

- **transferência com memória:**

LD Ri, v Carrega Registrador
 $R_i \leftarrow \text{Mem}[v]$: o conteúdo da palavra de memória de endereço v, é transferido para o registrador Ri.

ST v, Ri Armazena Registrador
 $\text{Mem}[v] \leftarrow R_i$: o conteúdo do registrador Ri é transferido para a palavra de memória de endereço v.

VLD Ri, v, Rj Carrega Registrador (endereçamento base)
 $R_i \leftarrow \text{Mem}[v + R_j]$: o conteúdo da palavra de memória de endereço igual a soma de v com o conteúdo do registrador Rj, é transferido para o registrador Ri.

VST v, Ri, Rj Armazena Registrador (endereçamento base)
 $\text{Mem}[v + R_j] \leftarrow R_i$: o conteúdo do registrador Ri é transferido para a palavra de memória de endereço igual a soma de v com o conteúdo do registrador Rj.

- **aritmética com ponto fixo:**

ICMP Ri, Rj, Flag Compara Inteiros
if Ri > Rj *then* Flag \leftarrow 0 *else if* Ri = Rj *then* Flag \leftarrow 2 *else* Flag \leftarrow 1: se o conteúdo do registrador Ri for maior, menor ou igual que o conteúdo de Rj, a instrução transfere para o indicador de condição Flag o valor 0, 1 ou 2. Em outras palavras, a instrução transfere para o indicador

de condição o valor 0, 1 ou 2 se o conteúdo do registrador R_i for respectivamente maior que, menor que, ou igual ao conteúdo do registrador R_j .

ILDI R_i , $iconst$ Carrega Imediato Inteiro
 $R_i \leftarrow iconst$: o conteúdo do registrador R_i torna-se igual a $iconst$.

IADD R_i , R_j , R_k Soma Inteiros
 $R_i \leftarrow R_j + R_k$: o conteúdo do registrador R_i torna-se igual a soma dos conteúdos dos registradores R_j e R_k .

ISUB R_i , R_j , R_k Subtrai Inteiros
 $R_i \leftarrow R_j - R_k$: o conteúdo do registrador R_i torna-se igual ao resultado da subtração do conteúdo do registrador R_k do conteúdo do registrador R_j .

IMUL R_i , R_j , R_k Multiplica Inteiros
 $R_i \leftarrow R_j * R_k$: o conteúdo do registrador R_i torna-se igual ao produto dos conteúdos dos registradores R_j e R_k .

IDIV R_i , R_j , R_k Divide Inteiros
 $R_i \leftarrow R_j \text{ div } R_k$: o conteúdo do registrador R_i torna-se igual ao quociente da divisão do conteúdo do registrador R_j pelo conteúdo do registrador R_k .

IINC R_i , $iconst$ Incrementa Inteiro
 $R_i \leftarrow R_i + iconst$: o conteúdo do registrador R_i é incrementado do valor de $iconst$.

IDEC R_i , $iconst$ Decrementa Inteiro
 $R_i \leftarrow R_i - iconst$: o conteúdo do registrador R_i é decrementado do valor de $iconst$.

● **aritmética com ponto flutuante:**

FCMP R_i , R_j , $Flag$ Compara em Ponto Flutuante
if $R_i > R_j$ *then* $Flag \leftarrow 0$ *else if* $R_i = R_j$ *then* $Flag \leftarrow 2$ *else* $Flag \leftarrow 1$.
 O efeito dessa instrução é semelhante ao da instrução ICMP. Podemos observar que ela compara operandos em ponto flutuante ao invés de operandos inteiros.

FLDI R_i , $fconst$ Carrega Imediato em Ponto Flutuante
 $R_i \leftarrow fconst$: o conteúdo do registrador R_i torna-se igual a $fconst$.

FADD R_i , R_j , R_k Soma em Ponto Flutuante
 $R_i \leftarrow R_j + R_k$: o conteúdo do registrador R_i torna-se igual a soma dos conteúdos dos registradores R_j e R_k .

FSUB R_i , R_j , R_k Subtrai em Ponto Flutuante
 $R_i \leftarrow R_j - R_k$: o conteúdo do registrador R_i torna-se igual ao resultado da subtração do conteúdo do registrador R_k do conteúdo do registrador R_j .

FMUL R_i , R_j , R_k Multiplica em Ponto Flutuante
 $R_i \leftarrow R_j * R_k$: o conteúdo do registrador R_i torna-se igual ao produto dos conteúdos dos registradores R_j e R_k .

FREC R_i , R_j Inverte em Ponto Flutuante
 $R_i \leftarrow 1 / R_j$: o conteúdo do registrador R_i torna-se igual ao inverso do conteúdo do registrador R_j .

• **lógicas:**

AND R_i , R_j , R_k E Lógico
 $R_i \leftarrow R_j \text{ and } R_k$: conteúdo do registrador R_i torna-se igual ao resultado do *and* lógico realizado entre os conteúdos dos registradores R_j e R_k .

OR R_i , R_j , R_k OU Lógico
 $R_i \leftarrow R_j \text{ or } R_k$: o conteúdo do registrador R_i torna-se igual ao resultado do *or* lógico realizado entre os conteúdos dos registradores R_j e R_k .

XOR R_i , R_j , R_k OU Exclusivo Lógico
 $R_i \leftarrow R_j \text{ xor } R_k$: o conteúdo do registrador R_i torna-se igual ao resultado do *or* exclusivo realizado entre os conteúdos dos registradores R_j e R_k .

NOT R_i , R_j NOT Lógico
 $R_i \leftarrow \text{not } R_j$: o conteúdo do registrador R_i torna-se igual ao resultado do *not* lógico realizado sobre o conteúdo dos registrador R_j .

- **de desvio:**

BRANCH Flag, PC+2, cond Desvio Condicional
if Flag = cond *then* PC ← PC+2: se o conteúdo do indicador de condição Flag corresponde a condição cond, então o fluxo de controle é desviado para o endereço PC+2.

JUMP endr Desvio Incondicional
 PC ← endr: o fluxo de controle é desviado para o endereço endr.

CALL endr Desvio para Subrotina
 salva PC; PC ← endr: o endereço de retorno é salvo, e o fluxo de controle é transferido para o endereço endr.

RET Retorna de Subrotina
 restaura PC: o fluxo de controle é dirigido para o endereço seguinte da instrução na qual a subrotina foi chamada.

- **transferência entre registradores:**

MOV Ri, Rj Transferência entre Registradores
 Ri ← Rj: o conteúdo do registrador Ri é transferido para o registrador Rj.

- **outras operações:**

NOP Não Opera
 nenhuma ação será executada.

HALT Para
 fim de execução de programa.

Além de gerar a forma intermediária, o compilador realiza algumas das principais otimizações normalmente presentes em compiladores comerciais, isto é, *constant folding*, eliminação de operações redundantes, substituições algébricas e eliminação de código morto ([AHOU86] e [FISC88]).

Durante a fase de otimização, o compilador realiza o *constant folding* em expressões. Isto é, expressões constantes são avaliadas em tempo de compilação. Por exemplo, a expressão:

(1) $x := 3 + 5 * 2;$

é transformada em:

(1) $x := 13;$

Na eliminação de operações redundantes, o compilador suprime no interior de cada bloco básico as instruções cuja execução não modifica o resultado produzido. Por exemplo, a seqüência:

(1) $a := b;$

(2) $b := a;$

(3) ...

é transformada em:

(1) $a := b;$

(2) ...

As substituições algébricas também são feitas pelo compilador no interior de blocos básicos e visam a troca de uma operação por outra semanticamente equivalente e computacionalmente mais eficiente. Por exemplo:

$x + 0 \implies x;$

$x - 0 \implies x;$

$x * 1 \implies x;$ e

$x / 1 \implies x.$

Finalmente durante a eliminação de código morto, o compilador elimina as instruções não alcançáveis pelo fluxo de controle do programa e aquelas que produzem um valor que não será mais utilizado ao longo do programa.

Embora o compilador não reconheça a estrutura de controle *Case*, que poderia ser útil em nossos experimentos sobre execução condicional, é importante lembrar que a escolha da forma intermediária deve-se ao seu reduzido número de instruções, tornando tratável o problema da execução condicional numa primeira etapa da nossa pesquisa.

Na Seção 3.3 apresentamos o conceito da execução condicional onde as ins-

truções contidas em uma instrução longa, são executadas condicional ou incondicionalmente.

3.3 A Execução Condicional

Processadores Super Escalares são caracterizados pela existência de múltiplas unidades funcionais independentes que podem ser ativadas durante cada ciclo de máquina, viabilizando dessa forma, a execução em paralelo de múltiplas instruções procedentes do mesmo programa de aplicação. A escolha das unidades que serão ativadas durante cada ciclo de máquina, é feita pelo algoritmo de escalonamento de instruções. Quando esse algoritmo é implementado por *software* temos além de outras, a arquitetura conhecida como VLIW (*Very Large Instruction Word*) [FISH83] e [FISH84a].

No nosso modelo básico de processador VLIW, as instruções presentes em uma instrução longa, podem ser executadas condicional ou incondicionalmente. Desse modo, podemos empacotar em uma mesma instrução longa, instruções provenientes de blocos básicos distintos, tornando assim, desnecessário introduzir no projeto da máquina mecanismos de predição de desvios e código de reparo ([SMIT81], [JLEE84], [LILJ88], [BUTL91], [HEST90] e [TAB91]).

Antes de apresentar a idéia da execução condicional, precisamos compreender o significado do termo “bloco sucessor” por nós empregado.

Um “bloco sucessor” (ou “bs”), é o bloco básico para onde o fluxo de controle é dirigido, quando na linguagem de alto nível temos uma estrutura do tipo IF *condição* THEN... (isto é, não existe ELSE correspondente ao THEN), e em tempo de execução, *condição* é verificada como sendo falsa.

A idéia fundamental é que durante o processo de compactação do código seqüencial, sejam empacotadas em uma mesma instrução longa, instruções oriundas de blocos básicos correspondentes as estruturas em linguagem de alto nível, do tipo “THEN e ELSE,” ou “THEN e o bloco sucessor”. Durante a ativação da instrução longa são executadas apenas as instruções cujas condições forem verdadeiras.

Para facilitar a apresentação, abreviamos os termos “bloco THEN” e “bloco ELSE” como “bt” e “be” respectivamente.

Para ilustrar o conceito de execução condicional, vamos examinar os trechos

de programas mostrados nas Figuras 3.1 e 3.2.

```

if (i and 1) = 1
  then odd := odd + 1
  else even := even + 1;

```

Figura 3.1: Um Trecho do Programa *Branch*

Na Figura 3.1 o bloco básico correspondente ao “bt” é composto pelo comando $odd := odd + 1$, e o correspondente ao “be” é composto pelo comando $even := even + 1$. Nesse caso, ao compactarmos em instruções longas, instruções provenientes dos dois blocos básicos examinados, é possível em tempo de execução, executar somente aquelas instruções cuja condição for satisfeita. Em outras palavras, se $(i \text{ and } 1) = 1$, as instruções provenientes do “bt” são executadas. Contudo se a condição for falsa então são executadas as instruções do “be.”

```

if ( (entr and mask ) <> 0 )
  then sum := sum + wgh;
  wgh := wgh * 2;

```

Figura 3.2: Um Trecho do Programa *BCD*

Já na Figura 3.2 o bloco básico correspondente ao “bt” é formado pelo comando $sum := sum + wgh$, ao passo que o “bs” é formado pelo comando $wgh := wgh * 2$. Dependendo da condição teremos em tempo de execução, instruções dos dois blocos sendo executadas simultaneamente, ou apenas as do bloco sucessor. Isto é, se $(entr \text{ and } mask) \neq 0$, então todas as instruções contidas na instrução longa são executadas. Se a condição for falsa, então apenas as instruções oriundas do “bs” são executadas. Podemos então concluir que as instruções do “bs” são sempre executada, ou seja, são executadas incondicionalmente.

Na Seção 3.4 descrevemos o modelo de processador Super Escalar utilizado nos nossos experimentos. Nesse modelo, as instruções contidas em uma instrução multifuncional, são executadas condicional ou incondicionalmente, de acordo com o especificado em cada instrução.

3.4 O Modelo de Arquitetura

O modelo de arquitetura capaz de executar os programas compactados da forma descrita, é composto basicamente pelos seguintes elementos:

- conjuntos de indicadores de condição (*flags*);
- um registrador de instrução;
- um banco de registradores;
- quatro tipos de unidades funcionais (UFs): unidade aritmética e lógica para inteiros (ALU), unidade para aritmética em ponto flutuante (FPU), unidade de acesso à memória (MEM) e unidade de processamento de desvios (BRANCH). O número de unidades funcionais de cada tipo é um parâmetro do modelo, contudo em qualquer caso, só há uma unidade de desvio.
- um apontador de instrução (PC); e
- barramentos conectando as UFs ao banco de registradores e aos conjuntos de indicadores de condição (*flags*).

A arquitetura suporta dois tipos de dados: inteiros e reais, ambos com largura de uma palavra de memória. Todas as vias de dados possuem também a largura de uma palavra. Somente palavras de memória são endereçáveis. Não é possível endereçar *bytes*.

A Figura 3.3 mostra as conexões existentes entre os diversos elementos do modelo.

Denominamos o nosso modelo de VLIW com capacidade de EXecução CONdi-cional. Para facilitar a apresentação, daqui em diante utilizaremos o termo CONDEX para denotar nosso modelo de execução condicional.

Para explorar mais eficientemente a concorrência do código seqüencial, decidimos fragmentar cada ciclo de processador do CONDEX em quatro subciclos. A leitura dos registradores fonte das operações é feita pelas unidades funcionais no primeiro subciclo. Da mesma forma as unidades funcionais só atualizam os registradores destino das operações no quarto subciclo. Isso permite por exemplo, que uma unidade funcional que use um registrador como operando fonte, obtenha seu

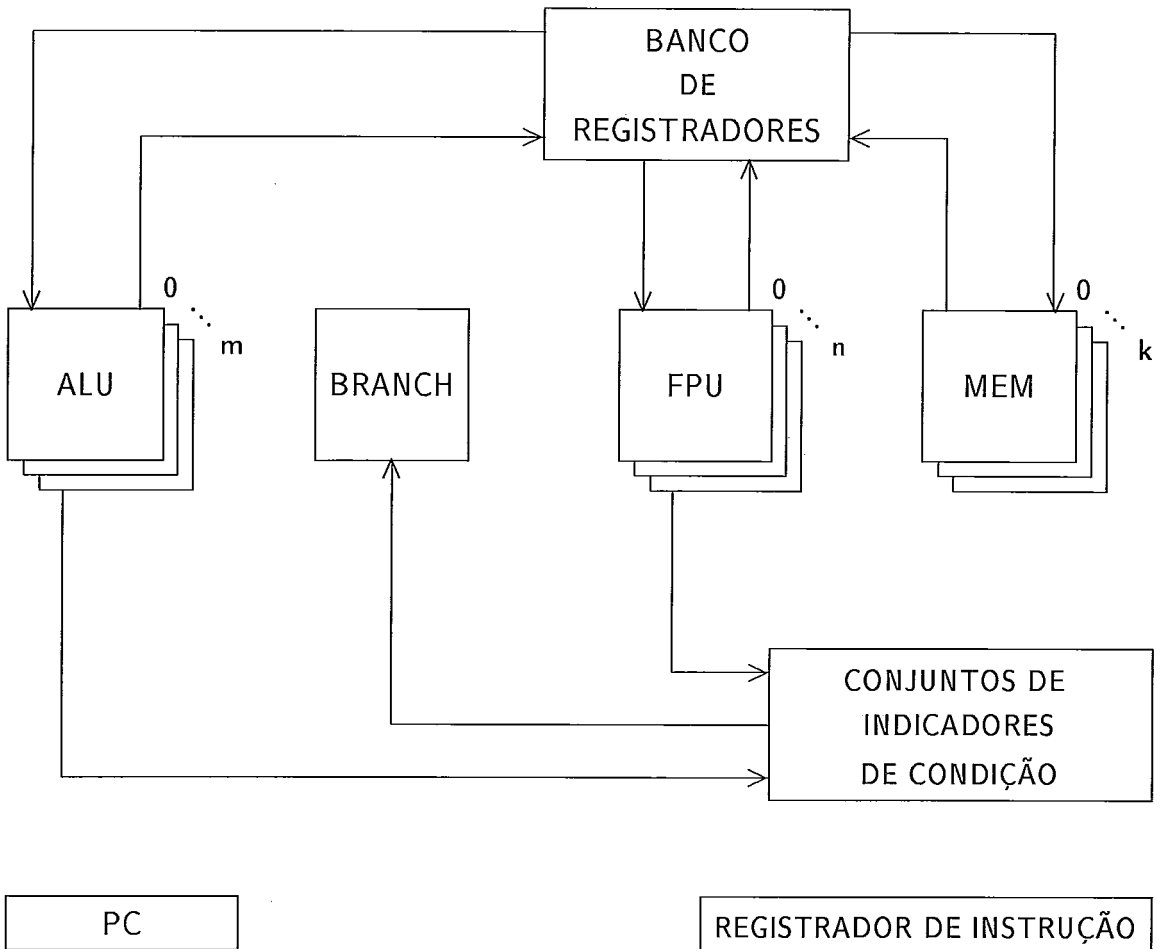


Figura 3.3: Esquema da Arquitetura Modelo – CONDEX

conteúdo no primeiro subciclo, possibilitando que o mesmo registrador possa ser usado como destino por uma outra unidade funcional (ou a mesma), já que o valor do registrador somente será modificado no último subciclo.

As unidades funcionais do modelo CONDEX, são capazes de executar cada uma das instruções geradas pelo compilador (descritas na Seção 3.2). Para não limitar o desempenho do modelo, alongando o ciclo de processador para a latência da instrução mais complexa, optamos por definir latências diferenciadas para as instruções.

O número de ciclos de máquina requeridos por cada instrução, é:

- LD, ST, VLD, VST: 4 ciclos.
- FMUL, FREC: 3 ciclos.

- IMUL, IDIV, FCMP, FADD, FSUB: 2 ciclos.
- ILDI, FLDI, ICMP, IADD, ISUB, IINC, IDEC AND, OR, XOR, NOT, BRANCH, JUMP, CALL, RET, MOV, NOP, HALT: 1 ciclo.

Como sabemos (vide Capítulo 2), uma instrução do código executável (ou instrução longa), é uma coleção de instruções de máquina que podem ser iniciadas concorrentemente. No nosso processador uma instrução longa consome um ciclo de máquina. Portanto se uma instrução iniciada em uma instrução longa demanda mais ciclos para ser executada, é preciso inserir NOPs nos correspondentes campos das instruções longas subseqüentes de modo a reservar a unidade funcional que a executa, por um número conveniente de ciclos.

Para exemplificar essa situação vamos considerar a instrução IMUL R5, R4, R3 que realiza a multiplicação de dois operandos inteiros, e cuja latência corresponde a dois ciclos do processador. Consideremos que a instrução é executada na ALU_i , que em seguida deve executar a instrução IINC R6, 1. Nesse caso, teremos seqüência de instruções longas mostrada na Figura 3.4.

ALU_i		
...	IMUL R5, R4, R3	...
...	NOP	...
...	IINC R6, 1	...

Figura 3.4: A Seqüência de Instruções para a ALU_i

Na Figura 3.4 vemos que uma instrução NOP foi inserida na instrução longa seguinte à que contém IMUL R5, R4, R3 afim de reservar a ALU_i por dois ciclos de máquina, relativos a latência da instrução IMUL.

3.5 Os Elementos do Modelo

Como vimos anteriormente, o modelo CONDEX inclui quatro tipos distintos de unidades funcionais e os seguintes elementos:

Conjuntos de Indicadores de Condição:

Um conjunto de indicadores de condição, é constituído por dois *flags* e possui largura de dois bits (um bit para cada *flag*). Os *flags* preservam seu conteúdo até que uma instrução de comparação (ICMP ou FCMP), os modifique. As instruções de comparação são implementadas (pelo *hardware* subjacente) através de uma subtração.

Mostramos um conjunto de indicadores de condição na Figura 3.5 com seus dois *flags* zero (Z) e negativo (N).

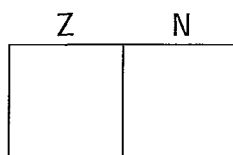


Figura 3.5: Um Conjunto de Indicadores de Condição

Os *flags* Z e N refletem o resultado da execução de uma instrução de comparação do seguinte modo:

- $Z = 1$ e $N = 0 \implies$ quando o resultado da subtração for nulo (igualdade);
- $Z = 0$ e $N = 0 \implies$ quando o resultado da subtração for positivo (maior que);
- $Z = 0$ e $N = 1 \implies$ quando o resultado da subtração for negativo (menor que).

Para ilustrar a ação das instruções ICMP e FCMP sobre um conjunto de indicadores de condição, tomemos como exemplo a comparação dos conteúdos dos registradores R4 e R5 que modifica o estado do conjunto de indicadores de condição 1. Ou seja a instrução:

ICMP R4, R5, 1

Considerando que os conteúdos dos registradores R4 e R5 é igual a 5, então ao término da execução da instrução, o conjunto de indicadores de condição 1 estará no estado mostrado na Figura 3.6. Isto é, como o resultado da subtração $R4 - R5$ é igual a zero, então o *flag* Z recebe o valor 1 e o *flag* N o valor zero.

Por outro lado, se o conteúdo de R4 for igual a 3 e o conteúdo de R5 igual a 5,

Z	N
1	0

Figura 3.6: Efeito da Execução de ICMP R4, R5, 1, se $R4 = 5$ e $R5 = 5$

então o conjunto 1 de indicadores estará no estado mostrado na Figura 3.7. Nesse caso como o resultado da subtração é negativo, então o *flag* Z é igual a zero, e o *flag* N igual a 1.

Z	N
0	1

Figura 3.7: Efeito da Execução de ICMP R4, R5, 1, se $R4 = 3$ e $R5 = 5$

Finalmente supondo que o conteúdo de R4 é igual a 5 e o de R5 igual a 3, então o efeito da comparação é mostrado na Figura 3.8. Nesse caso, o resultado da subtração $R4 - R5$ é positivo e ambos os *flags* do conjunto de indicadores de condição 1 são iguais a zero.

Z	N
0	0

Figura 3.8: Efeito da Execução de ICMP R4, R5, 1, se $R4 = 5$ e $R5 = 3$

A existência de um único conjunto de indicadores de condição força a execução do desvio condicional imediatamente após a avaliação da condição, caso contrário, os conteúdos dos *flags* poderiam ser alterados por uma outra instrução de comparação.

O modelo CONDEX inclui múltiplos conjuntos de indicadores de condição. A disponibilidade de múltiplos conjuntos de *flags* viabiliza que instruções provenientes das estruturas “bt-be” e “bt-bs” aninhadas, sejam compactadas em instruções multifuncionais comuns (isto é, IMFs que podem conter intruções dos diversos blocos

aninhados). Em outras palavras, se um outro conjunto de indicadores de condição for especificado, uma instrução de comparação aninhada ao ser executada, não destruirá o conteúdo dos *flags* modificados por uma instrução de comparação executada anteriormente.

O Registrador de Instrução:

O registrador de instrução do modelo, é capaz de armazenar um número de instruções igual ao de UFs da arquitetura. Na Figura 3.9, vemos o registrador de instrução com campos para as $n + 1$ ALUs, $m + 1$ FPUs, $k + 1$ MEMs e a única unidade funcional BRANCH do modelo.



Figura 3.9: Registrador de Instrução da Arquitetura Condicional

Além do código de operação e dos operandos, cada campo do registrador de instruções especifica quais as condições que devem ser satisfeitas para que a instrução seja executada. A Figura 3.10 mostra o diagrama de um campo do registrador de instruções.

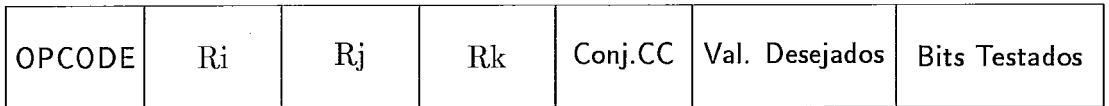


Figura 3.10: Um Campo do Registrador de Instrução

Na Figura 3.10, o campo “Conj. CC” especifica qual dos conjuntos de indicadores de condição deve ser considerado, para a execução da instrução. O campo “Val. Desejados”, especifica os valores que os *flags* (do conjunto de indicadores selecionado pelo campo Conj CC) devem conter para que a instrução seja executada. Finalmente, o campo “Bits Testados” indica os *flags* que devem ser testados. Se o campo “Bits Testados” for igual a zero, então a instrução é executada incondicionalmente. Os demais campos especificam o código da operação (OPCODE) e os operandos fonte (R_j e R_k) e destino (R_i).

O Banco de Registradores:

Os registradores do banco têm o mesmo número de bits (isto é têm a mesma

largura), que é igual a largura de uma palavra de memória. Eles podem armazenar indistintamente valores inteiros e em ponto-flutuante.

Todos os registradores são do tipo escrita-após-leitura, isto é, o valor do registrador é lido no início do ciclo de máquina (no primeiro subciclo), e reescrito no final (último subciclo).

As Unidades Funcionais:

- ALU é a unidade funcional que opera com inteiros, e é capaz de executar as instruções ICMP, ILDI, IADD, ISUB, IMUL, IDIV, IINC, IDEC, AND, OR, XOR, NOT, MOV e NOP.
- FPU é a unidade que opera com dados em ponto-flutuante, e é capaz de executar as instruções FCMP, FLDI, FADD, FSUB, FMUL, FREC, MOV e NOP.
- MEM é a unidade responsável pelas instruções de acesso a memória de dados e de programa. Ela é capaz de executar as instruções LD, ST, VLD, VST e NOP.
- BRANCH é a unidade que executa as instruções de desvio de um programa. Ela é capaz de executar as instruções BRANCH, JUMP, CALL, RET e NOP.

Uma configuração de arquitetura pode incluir uma ou mais ALUs, FPUs e unidades de memória (MEM), porém somente uma única unidade de desvio (BRANCH).

3.6 A Conexão Entre os Elementos do Modelo

A conexão entre os diversos elementos do CONDEX é realizada através dos bancos de registradores e de indicadores. Isto é, as saídas do banco de registradores e dos conjuntos de indicadores de condição são conectadas às entradas de unidades funcionais. Por outro lado, somente o banco de registradores e os conjuntos de indicadores de condição são conectados às saídas das unidades funcionais. Em outras palavras, uma unidade funcional sempre recebe os operandos e transfere os resultados das operações executadas, do/no banco de registradores ou do/no conjunto de indicadores de condição. A Figura 3.3 mostra a conexão dos elementos do CONDEX.

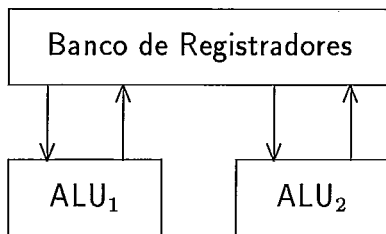


Figura 3.11: A Conexão de ALUs Através do Banco de Registradores

Para exemplificar a situação descrita acima, vamos considerar o caso em que uma ALU produz um resultado que deve ser usado como operando por uma outra ALU (ou até mesmo pela própria). A ALU não pode enviar o resultado da operação à outra ALU diretamente para que este seja utilizado como operando, este resultado deve obrigatoriamente, ser transferido para um registrador e posteriormente repassado à outra ALU (ou a mesma) para ser usado como operando. A Figura 3.11 mostra como é realizada a conexão de duas ALUs (ALU_1 e ALU_2) através do banco de registradores.

3.7 O Simulador do Modelo

Para avaliar os ganhos obtidos na paralelização do código seqüencial, observar o efeito da execução condicional, e coletar dados estatísticos que nos ajudem a chegar a uma configuração que apresente menor taxa “custo \times benefício,” especificamos e implementamos um simulador parametrizável do modelo.

Variando os parâmetros desse simulador, obtemos uma família de processadores derivados do modelo básico. Um componente dessa família difere de um outro pelo número de:

- ALUs;
- FPUs;
- MEMs;
- registradores;
- conjuntos de indicadores de condição.

O simulador ao interpretar um programa, mostra na tela do computador hospedeiro a instrução em execução, o conteúdo dos registradores, dos conjuntos de

indicadores de condição e o valor atual do PC. No fim da interpretação é possível listar o conteúdo das posições de memória, para a verificação dos valores finais das variáveis do programa interpretado.

Como veremos no Capítulo 5, três configurações distintas foram utilizadas durante nossos experimentos.

No Capítulo 4 a seguir examinamos o processo de paralelização do código seqüencial.

Capítulo 4

Compactação de Código

4.1 Introdução

O modelo CONDEX combina o paralelismo de baixo nível, explorado em processadores do tipo VLIW, com o conceito de execução condicional. Por esse motivo, o modelo permite escalonar instruções provenientes de diferentes blocos básicos¹ em uma mesma instrução multifuncional (ou IMF).

Nesse capítulo apresentaremos as fases de geração do código paralelo que será interpretado pelo simulador do modelo CONDEX. Descreveremos inicialmente o processo de compactação local do código intermediário seqüencial (descrito no Capítulo 3). Em seguida, apresentaremos a técnica de compactação condicional.

4.2 A Geração do Código Paralelo

O processo de geração do código paralelo compreende duas fases distintas. Na primeira, os blocos básicos do código intermediário seqüencial são identificados e cada um deles é submetido (individualmente) ao algoritmo de compactação local. Na segunda etapa, o programa compactado localmente (durante a primeira fase) é submetido à compactação condicional.

Conforme mencionado no Capítulo 2, técnicas de compactação local atuam somente no interior de um bloco básico. Experimentos descritos em [SHUS77],

¹Blocos Básicos são definidos como aqueles trechos de programa seqüencial, sem pontos de entrada, exceto no início do trecho, e sem comandos de desvio, exceto se for o último comando de trecho.

[GROS82], [MCFA86] e [DAVI90] constataram que blocos básicos geralmente são formados por um pequeno número de instruções. Por esse motivo, o paralelismo de baixo nível que pode ser extraído por algoritmos de compactação local é bastante limitado.

Por outro lado, as técnicas de compactação global atuam no programa como um todo (i.e., elas não levam em consideração as fronteiras delimitadas pelos blocos básicos). Conseqüentemente, elas são capazes de explorar mais eficientemente o paralelismo existente no código seqüencial: através das técnicas de compactação global, podemos alocar instruções provenientes de blocos básicos distintos numa mesma instrução multifuncional, o que viabiliza a execução antecipada de instruções.

O algoritmo de compactação condicional que especificamos durante nossos trabalhos pertence à classe das técnicas de compactação global. Ele visa paralelizar instruções provenientes das estruturas “bt e be” e “bt e bs” da linguagem Pascal. Essa geração de código paralelo é assim realizada:

- quando aplicado aos pares “bt e be,” o algoritmo de compactação condicional produz um único bloco de instruções multifuncionais (i.e., um bloco formado por instruções longas). Essas instruções multifuncionais contêm instruções dos dois blocos originais (i.e., dos blocos “bt” e “be”). Adicionalmente, o algoritmo explora a capacidade de execução condicional do modelo, especificando (em cada instrução multifuncional) quais as instruções que devem ser executadas. O algoritmo introduz nos campos da instrução longa, indicadores de condição que, em conjunto com o conteúdo dos *flags* da máquina, irão habilitar (ou não) a execução da instrução especificada pelo campo. Nesse caso, embora na IMF possam existir instruções pertencentes aos dois blocos básicos, em uma dada instrução multifuncional, serão executadas instruções pertencentes ao bloco “bt” ou ao bloco “be.”
- quando aplicada aos pares “bt e bs,” o algoritmo produz um único bloco com IMFs contendo instruções dos dois blocos. Como anteriormente, os componentes das instruções longas também possuem informações sobre as condições de execução. Nesse caso porém, as instruções do bloco “bt” somente serão executadas se a condição especificada pelo comando “IF” for verdadeira. Tal não ocorre para as instruções do bloco “bs.” As instruções desse bloco serão executadas independentemente do resultado

produzido pela avaliação do “IF.”

Na condução dos nossos experimentos com geração de código paralelo, a fase de compactação local é realizada automaticamente. Já a segunda fase, a compactação condicional, é realizada interativamente: compete ao usuário a tarefa de selecionar os blocos que serão compactados. Parte desse processo é levada a cabo manualmente.

4.2.1 O Processo de Compactação Local

O programa responsável pela compactação local (o Compactador Local) recebe como entrada o código intermediário seqüencial. Ele identifica os blocos básicos do programa e constrói o grafo do fluxo de controle correspondente.

O processo de compactação local é realizado separadamente para cada bloco básico do grafo de fluxo do programa. A partir das instruções de cada bloco básico, obtemos como resultado, uma seqüência de instruções multifuncionais. Durante o processo de compactação local, os campos “*Conj. CC*”, “*Val. Desejados*”, “*Bits Testados*” (descritos no Capítulo 3), são ignorados.

O objetivo da primeira fase da geração de código é tão somente compactar (em instruções longas) aquelas instruções do bloco básico que podem ser executadas simultaneamente, explorando desse modo, o paralelismo de baixo nível existente no interior do bloco básico. Os campos das IMFs que viabilizam a execução condicional serão preenchidos posteriormente, na segunda fase.

O Compactador Local, procura movimentar instruções uma a uma, para cima, tentando grupá-las em instruções longas. Utilizamos uma versão do algoritmo *list-scheduling* [LAND80] para compactar as instruções dos blocos básicos. Esse algoritmo é controlado por parâmetros e foi adaptado para o nosso modelo básico de máquina. O número e tipo de unidades funcionais, e o tempo de latência das diferentes operações, são exemplos de parâmetros que influenciam o processo de compactação de código. A movimentação de instruções leva em conta as dependências de dados, os conflitos na utilização dos recursos presentes na configuração da máquina e o tempo de latência das unidades funcionais. No final da compactação, o algoritmo garante que o bloco resultante é semanticamente equivalente ao bloco original.

O processo de compactação tira proveito de uma importante característica in-

corporada no modelo CONDEX: instruções anti-dependentes podem compartilhar uma mesma IMF. Essa característica aumenta o grau de preenchimento das IMFs (e conseqüentemente o nível de paralelismo durante a execução dos programas de aplicação). A presença de instruções anti-dependentes na mesma IMF é viabilizada pelo esquema de temporização adotado no modelo. Conforme mencionado no Capítulo 3, os registradores que atuarão como fonte das operações são lidos no primeiro subciclo do processador e somente no último subciclo é que os registradores destino são atualizados. Desse modo, garantimos que os conteúdos dos registradores não serão alterados por instruções apresentando anti-dependências. A seguir, apresentamos uma breve descrição do processo de compactação local.

No início do processo de compactação local, associamos a cada instrução do bloco básico uma instrução multifuncional. Em seguida, cada instrução é movida para cima, de IMF em IMF, até que:

- ocorra conflito na utilização de recursos na IMF destino; ou
- ocorra uma dependência de dados (verdadeira ou de saída), com outras instruções já escalonadas na IMF destino.

Em outras palavras, o Compactador Local recebe como entrada um bloco básico B formado por uma seqüência de instruções longas. Inicialmente, cada IMF de B contém uma única instrução. O algoritmo produz como saída um novo bloco básico compactado. Durante a compactação, o algoritmo leva em consideração a latência das instruções que estão sendo movimentadas. Por exemplo, vamos supor que I_1, I_2, \dots, I_n é a seqüência de instruções do bloco básico de entrada. Precisamos lembrar que cada instrução I_i é formada pela instrução propriamente dita, seguida dos NOPs correspondentes a sua latência (vide Seção 3.4). Portanto, ao tentar movimentar a instrução para uma IMF precedente, temos que reservar a unidade funcional correspondente durante o número de ciclos requeridos para a execução da instrução. A introdução de instruções do tipo NOP nos campos apropriados das IMFs sucessoras, garante essa reserva.

Finalmente, após escalonar as instruções de cada bloco básico do programa, o Compactador Local atualiza o grafo de fluxo de controle, modificando os endereços das instruções de desvio de modo a refletir os novos endereços do programa paralelizado.

4.2.2 O Processo de Compactação Condicional

O processo de compactação condicional é levado a cabo interativamente. Ele utiliza como entrada os blocos básicos gerados pelo Compactador Local e produz como resultado um programa objeto paralelo. Inicialmente, as estruturas “bt e be” (i.e., pares de blocos básicos “THEN” e “ELSE” provenientes do comando “IF” da linguagem Pascal) e “bt e bs” (pares de blocos básicos provenientes de um comando “IF” sem a cláusula “ELSE”) são identificadas. Em seguida, o algoritmo de compactação condicional tentará alocar num único bloco de IMFs, instruções pertencentes aos blocos básicos de cada par. Isto é, sempre que for possível, instruções do bloco “be” serão escalonadas nas IMFs contendo instruções do bloco “bt,” e as instruções do bloco “bs” serão alocadas nas instruções multifuncionais contendo instruções do bloco “bt.” Um esboço dos passos realizados durante o processo de compactação condicional está ilustrado nas Figuras 4.1 a 4.4. Para facilitar a apresentação dessas figuras, o termo bloco básico foi abreviado para “bb.”

A primeira fase do algoritmo de compactação condicional realiza a identificação dos pares de blocos “bt e be” e “bt e bs”. A Figura 4.1 ilustra essa fase.

```

1. for i = 1 to penúltimo bb do programa do
    if bbi é um bb ainda não examinado and bbi é um bloco do tipo bt
        then if ∃ o bloco be correspondente
            then begin
                marca o par de blocos como bt-be;
                marca o bloco be como já examinado;
            end
        else if ∃ o bloco bs correspondente
            then begin
                marca o par de blocos como bt-bs;
                marca o bloco bs como já examinado;
            end;

```

Figura 4.1: Identificação dos Pares “bt-be” e “bt-bs”

Após a identificação dos pares “bt e be” e “bt e bs,” o algoritmo realiza a fusão dos pares que foram marcados durante a primeira fase. As Figuras 4.2 e 4.3

ilustram respectivamente, o escalonamento de instruções do “be” em IMFs do “bt” e de instruções do “bs” em instruções longas do “bt.”

```

2. while  $\exists$  par bt-be do
  begin
    i := 1;                                     {i controla o n.º de IMFs do bt}
    for j = 1 to n.º de IMFs do be do          {j controla o n.º de IMFs do be}
      while  $\exists$  instrução na IMFj do
        begin
          for k = 1 to n.º de instruções na IMFj do {k controla o n.º de instr. na IMFj}
            if instruçãok  $\neq$  NOP
              then begin
                if instruçãok não causa conflito de recursos na IMFi do
                  bt and por todos os n ciclos de latência da instruçãok não
                  causa conflito de recursos com IMFi+1, ... , IMFi+n do bt
                  then begin
                    move instruçãok para IMFi do bt;
                    reserva a unidade funcional correspondente por n
                    ciclos;
                    introduz máscara no campo apropriado da IMFi;
                  end;
                end;
              end;
            i := i + 1;
            if i > n.º de IMFs do bt
              then goto 20;
          end;
20: if  $\exists$  IMF vazia no be and  $\nexists$  UF reservada nesse ciclo
    then elimina IMF;
    if endereço da última IMF de be > endereço da última IMF de bt or endereço da
    última IMF de bt > endereço da última IMF de be
      then introduz instrução JUMP na IMF de endereço menor para saltar IMFs do
      bloco maior;
    elimina as duas IMFs anteriores ao bt que contêm as instruções BRANCH e JUMP;
  end;

```

Figura 4.2: Segunda Fase do Algoritmo – Fusão dos Pares “bt-be”

```

3. while  $\exists$  par bt-bs do
  begin
    i := 1;                                {i controla o n.º de IMFs do bt}
    for j = 1 to n.º de IMFs do bs do      {j controla o n.º de IMFs do bs}
      while  $\exists$  instrução na IMFj do
        begin
          for k = 1 to n.º de instruções na IMFj do {k controla o n.º de instr. na IMFj}
            if instruçãok  $\neq$  NOP
              then begin
                if instruçãok não causa conflito de recursos na IMFi do
                  bt and a instruçãok não causa conflito de dados com IMFi
                  do bt and por todos os n ciclos de latência da instruçãok não
                  causa conflito de recursos com IMFi+1, ... , IMFi+n, do bt
                  then begin
                    move instruçãok para IMFi do bt;
                    reserva a unidade funcional correspondente por n
                    ciclos;
                    introduz máscara no campo apropriado da IMFi;
                  end;
                end;
              end;
            i := i + 1;
            if i > n.º de IMFs do bt
              then goto 30;
          end;
30: if  $\exists$  IMF vazia no bs and  $\nexists$  UF reservada nesse ciclo
      then elimina IMF;
      if endereço da última IMF de bs < endereço da última IMF de bt
        then introduz instrução JUMP na última IMF de bs para saltar IMFs do bt;
      elimina as duas IMFs anteriores ao bt que contêm as instruções BRANCH e JUMP;
    end;
  end;

```

Figura 4.3: Terceira Fase do Algoritmo – Fusão dos Pares “bt-bs”

Na quarta fase os endereços das instruções de desvio são corrigidos (vide Figura 4.4).

4. Atualiza endereços das instruções de desvio de todo o programa compactado;

Figura 4.4: Quarta Fase do Algoritmo – Atualização dos Endereços

Durante o processo de compactação condicional, o algoritmo movimenta as instruções para cima (uma instrução por vez), tentando grupá-las em instruções longas parcialmente preenchidas pelo processo de compactação local. Analogamente ao que ocorre durante a compactação local, o processo de compactação condicional leva em conta as dependências de dados (verdadeiras e de saída) e os conflitos na utilização de recursos.

Após a fusão dos pares “bt-be” e “bt-bs,” o algoritmo de compactação condicional preenche os campos “*Conj. CC,*” “*Val. Desejados*” e “*Bits Testados*” convenientemente, remove as IMFs que tornaram-se vazias, e atualiza os endereços de desvio.

Se for vantajoso, o algoritmo pode introduzir instruções de desvio incondicional para transferir o fluxo de controle diretamente para IMFs relacionadas com os blocos “be” ou “bs.” Isso ocorre toda vez que o algoritmo detecta que parte do bloco resultante da fusão é formada somente por instruções pertencentes a um dos blocos. Nesse caso específico, a introdução de uma instrução de desvio evitará a busca daquelas IMFs contendo apenas instruções do bloco “THEN” (ou do bloco “ELSE”) e que não serão executadas se a condição do comando “IF” for falsa (ou verdadeira).

Apresentamos a seguir, três exemplos que ilustram o processo de compactação condicional. Para cada exemplo, mostramos o trecho do programa fonte, o grafo do fluxo de controle (GFC) correspondente, os blocos básicos gerados pelo processo de compactação local e os que foram produzidos pelo algoritmo de compactação condicional. O primeiro exemplo apresenta o efeito da compactação condicional em um par “bt e bs.” Os outros dois exemplos abordam a fusão de pares “bt e be.” Em particular, o terceiro exemplo mostra a introdução de comandos de desvio incondicional naqueles casos em que é vantajoso saltar (em tempo de execução) IMFs contendo somente instruções de um dos blocos do par.

Nas figuras relacionadas com o grafo do fluxo de controle dos programas, cada nó do GFC representa um bloco básico já compactado localmente. No interior de cada nó do grafo, estão indicados os endereços das IMFs que fazem parte do bloco

básico, e na parte externa identificamos o tipo do bloco (THEN, ELSE ou BS). As IMFs envolvidas no processo de compactação condicional, estão localizadas no interior da área delimitada pela linha tracejada do grafo do fluxo.

Nas figuras contendo o trecho compactado localmente e nas figuras com o código resultante da compactação condicional, os blocos básicos estão no interior de retângulos tracejados e cada instrução NOP é representada pelo caractere “—”. Os endereços das IMFs estão no lado esquerdo dos retângulos e no lado direito temos o número do bloco básico e a identificação do seu tipo (THEN, ELSE ou BS).

No primeiro exemplo, estamos assumindo que a configuração derivada do modelo CONDEX inclui as seguintes unidades funcionais: 1 ALU, 1 FPU, 2 MEMs e 1 BRANCH. É importante observar que a FPU da configuração não é usada pelo programa exemplo. Por esse motivo, a FPU foi omitida nas figuras.

```

if x[k] < x[m]
  then m := k;
  k := k + 1;

```

Figura 4.5: Trecho do Programa *Livermore Loop 24*

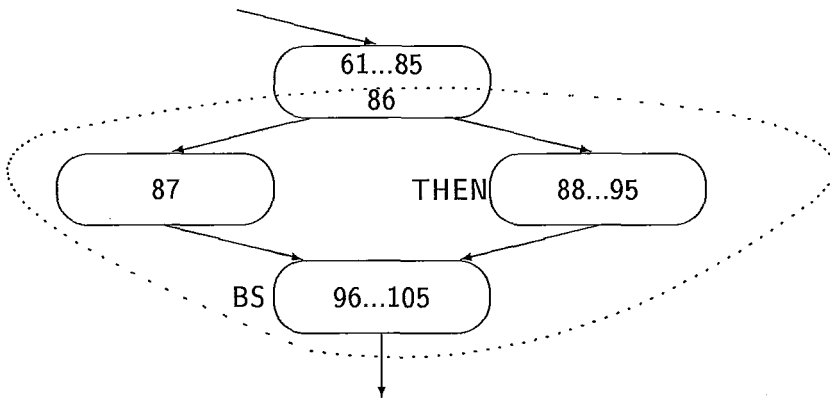


Figura 4.6: GFC do Trecho do Programa *Livermore Loop 24*

As quatro primeiras figuras referem-se ao programa *Livermore Loop número 24*. Abordaremos somente o trecho do programa listado na Figura 4.5. Este trecho corresponde aos blocos “bt e bs” do grafo do fluxo de controle da Figura 4.6.

A Figura 4.7 apresenta o código resultante da compactação local e a Figura 4.8 mostra o código compactado condicionalmente.

Na Figura 4.5, o bloco básico “bt” corresponde ao comando $m := k$, enquanto que o bloco “bs” contém o comando $k := k + 1$. As Figuras 4.6 e 4.7 mostram os blocos básicos e as IMF’s de cada bloco. O algoritmo de compactação local empacotou as instruções do programa em quatro blocos básicos. Após o processo de compactação condicional (vide Figura 4.8), o programa ficou reduzido a dois blocos básicos.

	ALU ₀	MEM ₀	MEM ₁	BRANCH	
•	•	•	•	•	bb1
•	•	•	•	•	
•	•	•	•	•	
85	ICMP R2,R7,1	—	—	—	bb2
86	—	—	—	BRANCH 1,88	
87	—	—	—	JMP 96	T H E N
88	—	—	VLD R8,-2,R1	—	
89	—	—	—	—	
90	—	—	—	—	
91	—	—	—	—	
92	—	—	VST -1,R1,R8	—	
93	—	—	—	—	
94	—	—	—	—	
95	—	—	—	—	bb4
96	ILD1 R4,1	—	VLD R9,-2,R1	—	
97	—	—	—	—	
98	—	—	—	—	
99	—	—	—	—	
100	IADD R5,R9,R4	—	—	—	
101	—	—	VST -2,R1,R5	—	B S
102	—	—	—	—	
103	—	—	—	—	
104	—	—	—	—	
105	—	—	—	JMP 54	

Figura 4.7: Trecho do Programa *Livermore Loop 24* após Compactação Local

Examinando as Figuras 4.7 e 4.8, podemos constatar que o trecho de programa compactado localmente é formado por 21 IMF’s e que após a compactação condicional, ele ficou somente com 11 instruções longas.

	ALU ₀	MEM ₀	MEM ₁	BRANCH	
•	•	•	•	•	bb1
•	•	•	•	•	
•	•	•	•	•	
85	ICMP R2,R7,1,0,0,0	—	—	—	
86	ILDI R4,1,0,0,0	VLDR9,-2,R1,0,0,0	VLDR8,-2,R1,1,1,1	—	bb2
87	—	—	—	—	
88	—	—	—	—	
89	—	—	—	—	T H E N + B S
90	IADD R5,R9,R4,0,0,0	—	VST -1,R1,R8,1,1,1	—	
91	—	VST -2,R1,R5,0,0,0	—	—	
92	—	—	—	—	
93	—	—	—	—	
94	—	—	—	—	
95	—	—	—	JMP 54,0,0,0	

Figura 4.8: Trecho do Programa *Livermore Loop 24* após Compactação Condicional

Conforme mostrado na Figura 4.8, foram acrescentados três novos campos em cada instrução das IMFs. Esses campos correspondem aos valores de “*Conj. CC*,” “*Val. Desejados*” e “*Bits Testados*” (as máscaras). Podemos notar também que esses valores são “1, 1, 1” para as instruções do bloco “bt” e “0, 0, 0” para as instruções do “bs.” As instruções do bloco “bs” terão que ser executadas incondicionalmente: sua execução independe da avaliação do comando “IF.” Por esse motivo, o algoritmo de compactação condicional introduziu zeros no campo de máscara de cada instrução do bloco “bs” (se o campo “*Bits Testados*” for igual a zero, nenhum bit de qualquer conjunto de indicadores de condição necessita ser testado para que a instrução seja executada). Por outro lado, as instruções do bloco “THEN” somente serão executadas se a condição do comando “IF” for verdadeira. Ao encontrar o campo da máscara com o valor “1, 1, 1,” o processador verificará se o bit menos significativo do primeiro conjunto de indicadores de condição é igual a “1.” Se for o caso, a instrução é executada.

Na Figura 4.8 observamos ainda que o endereço da instrução de desvio incondicional “JUMP 54” (localizada na IMF 95) não foi alterado. Esse endereço está localizado antes dos blocos que sofreram a compactação condicional, isto é, trata-se de um desvio para um endereço mais baixo e portanto não precisa ser modificado.

O segundo exemplo que vamos abordar refere-se ao trecho do programa *Branch* (vide Figura 4.9). Nesse trecho temos um bloco “bt” e um “be.” Estamos assumindo que a configuração do processador inclui 2 ALUs, 1 FPU, 2 MEMs e 1 BRANCH. Como anteriormente, a única FPU da arquitetura não é utilizada e por esse motivo ela foi omitida. A Figura 4.10 mostra o GFC do trecho de programa listado na Figura 4.9.

```

if (i and 1) = 1
  then odd := odd + 1
  else even := even + 1;

```

Figura 4.9: Trecho do Programa *Branch*

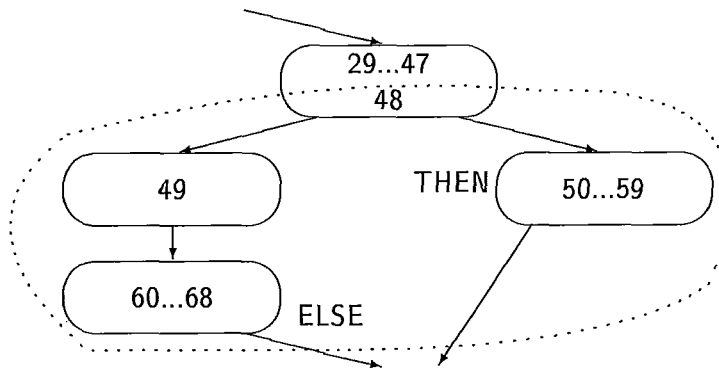


Figura 4.10: GFC do Trecho do Programa *Branch*

As Figuras 4.11 e 4.12 apresentam respectivamente o código resultante da compactação local e da condicional. Analogamente ao que ocorreu com o exemplo anterior, os quatro blocos básicos foram reduzidos a dois pelo algoritmo de compactação condicional. Podemos observar também, a marcante redução no tamanho do código estático: o trecho de programa foi reduzido de 22 para 10 IMFs pelo processo de compactação condicional.

	ALU ₀	ALU ₁	MEM ₀	MEM ₁	BRANCH	
•	•	•	•	•	•	bb1
•	•	•	•	•	•	
•	•	•	•	•	•	
47	ICMP R2,R3,3	—	—	—	—	
48	—	—	—	—	BRANCH 3,50	
49	—	—	—	—	JMP 60	bb2
50	ILDI R4,1	—	—	VLD R3,-2,R1	—	bb3
51	—	—	—	—	—	
52	—	—	—	—	—	
53	—	—	—	—	—	T H E N
54	IADD R5,R3,R4	—	—	—	—	
55	—	—	—	VST -2,R1,R5	—	
56	—	—	—	—	—	
57	—	—	—	—	—	
58	—	—	—	—	—	
59	—	—	—	—	JMP 69	
60	ILDI R6,1	—	—	VLD R7,-3,R1	—	bb4
61	—	—	—	—	—	
62	—	—	—	—	—	
63	—	—	—	—	—	E L S E
64	IADD R8,R7,R6	—	—	—	—	
65	—	—	—	VST -3,R1,R8	—	
66	—	—	—	—	—	
67	—	—	—	—	—	
68	—	—	—	—	—	

Figura 4.11: Trecho do Programa *Branch* após Compactação Local

Durante o processo de compactação condicional, o algoritmo eliminou as instruções de desvio condicional, provocando a fusão de blocos. Por exemplo, conforme ilustrado nas Figuras 4.11 e 4.12, o algoritmo de compactação condicional removeu as IMFs 48 e 49. A primeira continha a instrução (“BRANCH 3, 50”) que transferia o controle para o bloco “THEN” se a condição do comando “IF” fosse verdadeira. Analogamente, a IMF de endereço 49 que transferia o controle para o bloco “ELSE” também foi removida. O conceito de execução condicional torna redundante essas instruções de desvio: o campo de máscara de cada instrução indica se ela deve ser executada ou não. Nesse exemplo, a máscara especifica que o terceiro conjunto de *flags* será usado para controlar a execução das instruções oriundas do bloco “be” ou

do bloco “bt.” É importante observar que a condição do comando “IF” será avaliada pela instrução de comparação “ICMP R2, R3, 3.” Essa instrução (vide IMF de endereço 47 na Figura 4.12) compara os registradores R2 e R3, armazenando o resultado no terceiro conjunto de *flags* do processador.

	ALU ₀	ALU ₁	MEM ₀	MEM ₁	BRANCH	
•	•	•	•	•	•	bb1
•	•	•	•	•	•	
•	•	•	•	•	•	
47	ICMP R2,R3,3,0,0,0	—	—	—	—	
48	ILDI R4,1,3,2,2	ILDI R6,1,3,0,2	VLD R7,-3,R1,3,0,2	VLD R3,-2,R1,3,2,2	—	bb2
49	—	—	—	—	—	T
50	—	—	—	—	—	H
51	—	—	—	—	—	E
52	IADD R5,R3,R4,3,2,2	IADD R8,R7,R6,3,0,2	—	—	—	N
53	—	—	VST -3,R1,R8,3,0,2	VST -2,R1,R5,3,2,2	—	+
54	—	—	—	—	—	E
55	—	—	—	—	—	L
56	—	—	—	—	—	S
						E

Figura 4.12: Trecho do Programa *Branch* após Compactação Condicional

No trecho de programa compactado condicionalmente (vide Figura 4.12), a máscara das instruções relacionadas com o bloco “ELSE” é igual a “3, 0, 2.” Em tempo de execução, esse valor é assim interpretado: se o bit mais significativo (valor 2) do terceiro conjunto de *flags* for igual a zero (*false = 0*), a instrução é executada; caso contrário, ela é ignorada. Já para as instruções oriundas do bloco “THEN,” o valor da máscara é igual a “3, 2, 2.” Durante a execução do programa, essas instruções somente serão executadas se o bit mais significativo do terceiro conjunto de *flags* for igual a “1.” Desse modo, a execução de cada instrução nas IMFs do bloco “THEN+ELSE” é condicionada ao valor do bit mais significativo do terceiro conjunto de *flags* da máquina.

O terceiro exemplo mostra um trecho de programa cujas instruções do bloco “be” também foram empacotadas nas IMFs do bloco “bt.” O exemplo difere do anterior pelo número de IMFs do bloco “bt” que é bem menor do que o bloco “be.” Nesse caso, torna-se mais vantajoso introduzir uma instrução de desvio após a última instrução do bloco “bt,” saltando as IMFs contendo tão somente instruções

provenientes do bloco “be.” Se tal providência não fosse tomada, as fases de busca, decodificação e ativação das IMFs restantes (i.e., contendo somente instruções provenientes do bloco “ELSE”) aumentariam o tempo de execução do programa toda vez que a condição do comando “IF” fosse verdadeira (nessa situação, o restante do bloco seria interpretado como NOPs, introduzindo, desnecessariamente, um *overhead* no tempo de processamento).

Nesse exemplo, utilizaremos o trecho do programa *Árvore* listado na Figura 4.13. A Figura 4.14 mostra o grafo do fluxo de controle desse trecho de programa. Abordaremos somente as transformações produzidas nos quatro blocos básicos delimitados pela linha tracejada da Figura 4.14.

```

If vetorValor[No] = Param
  then achou := true
  else begin
    ant := no;
    if VetorValor[no] > Param Then No := FilhoEsq[No]
    else No := filhoDir[No];
  end;

```

Figura 4.13: Trecho do Programa *Árvore*

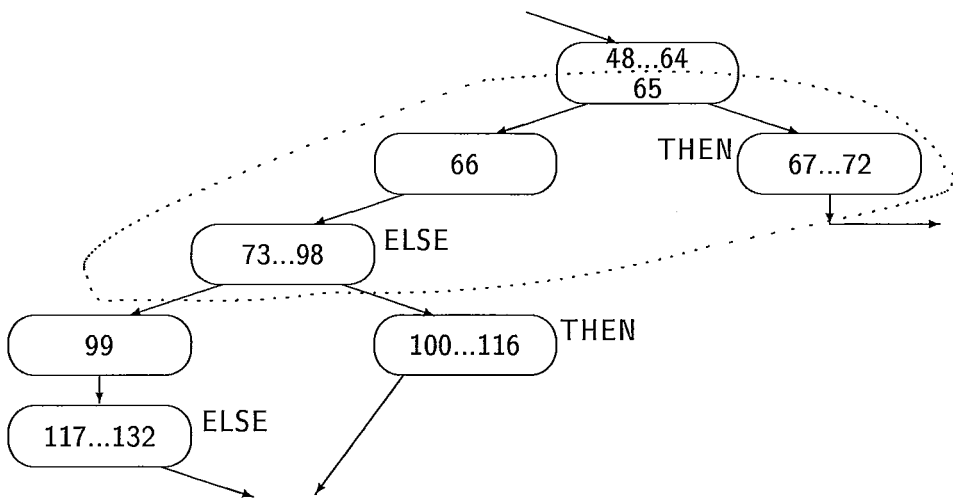


Figura 4.14: GFC do Trecho do Programa *Árvore*

No trecho da Figura 4.13, o bloco “bt” é formado pelo comando *achou := true*, e o bloco “be” pelos comandos *ant := no; if VetorValor...etc*. No processo de compactação, utilizamos uma configuração com as seguintes unidades funcionais: 2 ALUs, 1 FPU, 2 MEMs e 1 BRANCH. Tendo em vista que a FPU também não foi utilizada pelo trecho do programa, ela será omitida nas figuras.

Examinando as Figuras 4.15 e 4.16, podemos constatar que as 19 IMFs do código compactado localmente foram reduzidas a 7 IMFs pelo algoritmo de compactação condicional.

	ALU ₀	ALU ₁	MEM ₀	MEM ₁	BRANCH	
•	•	•	•	•	•	bb1
•	•	•	•	•	•	
•	•	•	•	•	•	
64	ICMP R5,R6,3	—	—	—	—	
65	—	—	—	—	BRANCH 3,67	
66	—	—	—	—	JMP 73	bb2
67	ILDI R7,1	—	—	—	—	bb3
68	—	—	—	VST -605,R1,R7	—	
69	—	—	—	—	—	T
70	—	—	—	—	—	H
71	—	—	—	—	—	E
72	—	—	—	—	JMP 34	N
73	ILDI R3,1	—	—	VLD R0,-1,R2	—	bb4
74	—	—	—	—	—	
75	—	—	—	—	—	
76	—	—	—	—	—	
77	—	—	—	VST -2,R2,R0	—	
78	—	—	—	—	—	E
79	—	—	—	—	—	L
80	—	—	—	—	—	S
81	—	—	—	VLD R3,-1,R2	—	E
82	—	—	—	—	—	
•	•	•	•	•	•	
•	•	•	•	•	•	
•	•	•	•	•	•	

Figura 4.15: Trecho do Programa *Árvore* após Compactação Local

Conforme mostrado na Figura 4.15, o terceiro conjunto de *flags* controla a

ativação do bloco “bt” ou do “be” (a IMF 65 transfere o controle para o bloco “THEN” e a IMF 67 para o bloco “ELSE”). Esse terceiro conjunto de indicadores de condição é modificado pela instrução “ICMP R5, R6, 3” localizada na IMF de endereço 64.

No código compactado localmente, a IMF de endereço 65 contém a instrução “BRANCH 3, 67” (vide Figura 4.15) que desvia para o bloco “THEN” se a condição for verdadeira. No código produzido pelo algoritmo de compactação condicional (vide Figura 4.16), essa instrução foi removida. O mesmo ocorreu com a instrução “JUMP 73” (que desvia para o bloco “ELSE”) na IMF de endereço 66.

	ALU ₀	ALU ₁	MEM ₀	MEM ₁	BRANCH	
•	•	•	•	•	•	bb1
•	•	•	•	•	•	
•	•	•	•	•	•	
64	ICMP R5,R6,3,0,0,0	—	—	—	—	
65	ILDI R7,1,3,2,2	ILDI R3,1,3,0,2	—	VLD R0,-1,R2,3,0,2	—	bb2
66	—	—	VST -605,R1,R7,3,2,2	—	—	T H E N +
67	—	—	—	—	—	
68	—	—	—	—	—	
69	—	—	—	VST -2,R2,R0,3,0,2	—	
70	—	—	—	—	JMP 34,3,2,2	
•	•	•	•	—	•	E L S E
•	•	•	•	—	•	
•	•	•	•	•	•	

Figura 4.16: Trecho do Programa *Árvore* após Compactação Condicional

Por outro lado, a última instrução do bloco “bt,” a instrução “JMP 34,” não foi eliminada pelo algoritmo de compactação condicional (vide Figura 4.16). Ocorre que é mais eficiente transferir o controle para o início do *loop*, evitando assim que as IMFs do bloco “be” sejam examinadas se a condição do “IF” for verdadeira.

No Capítulo 5 a seguir, apresentaremos os resultados da avaliação da qualidade do código produzido pelos algoritmos de compactação local e condicional. A avaliação considera o tamanho do código estático e apresenta algumas características dinâmicas dos programas de teste no simulador do modelo CONDEX.

Capítulo 5

Avaliação de Desempenho

5.1 Introdução

Este capítulo descreve a metodologia empregada durante nossos experimentos. Inicialmente, apresentamos as diferentes configurações do simulador CONDEX e a bateria de programas de teste utilizadas na avaliação de desempenho. Em seguida, mostramos como a compactação condicional reduziu o número de instruções e de IMFs em cada programa de teste, para cada configuração derivada do modelo CONDEX. Posteriormente, apresentamos os resultados obtidos (dinamicamente) durante a interpretação dos programas de teste nas diferentes configurações do nosso modelo. Finalmente, mostramos um resumo da avaliação realizada.

5.2 A Metodologia

Para avaliar a qualidade do código paralelo gerado pelo algoritmo de compactação condicional, selecionamos um conjunto de programas de teste. Esses programas foram compilados e compactados localmente. Em seguida, o código resultante da compactação local foi processado pelo algoritmo de compactação condicional, produzindo o código objeto derivado de cada programa da bateria de teste. Tendo em vista que foram utilizadas diferentes configurações de máquina, repetimos os procedimentos de compactação local e condicional para cada configuração. Em outras palavras, a partir do mesmo programa de teste, diferentes programas objeto foram gerados pelos procedimentos de compactação. Finalmente, os programas compactados foram interpretados pelas configurações do interpretador do modelo

CONDEX.

5.3 As Configurações da Arquitetura

Durante nossos experimentos, três configurações derivadas do modelo CONDEX foram empregadas. O número e mistura de unidades funcionais caracteriza cada configuração. Começamos com uma configuração que inclui um número reduzido de unidades funcionais (2 ALUs, 1 FPU, 1 MEM e 1 BRANCH) e variando o número de cada tipo de unidade funcional (exceto o da unidade funcional BRANCH), segundo potências de 2, obtivemos as outras configurações. Desse modo, usamos configurações com o número de unidades funcionais, registradores e conjuntos de indicadores de condição especificados na Tabela 5.1.

CONFIGURAÇÃO	ALUs	FPU _s	MEM _s	BRANCH	Registradores	Conj. Ind. Cond.
1	2	1	1	1	16	4
2	4	2	2	1	16	4
3	8	4	4	1	16	4

Tabela 5.1: As Configurações da Arquitetura do Modelo CONDEX

5.4 Os Programas Teste

A bateria de teste usada em nossos experimentos consiste de cinco programas. Os programas foram escolhidos por terem sido utilizados anteriormente em experimentos realizados na COPPE/UFRJ ([FERN92a], [FERN92b], [BARB93], [SOUZ93] e [MONT93]) e em outras universidades ([MCMA83], [DITZ87] e [AUHT85]). Adicionalmente, a manipulação de programas simples, facilita o processo de compactação interativo. O código fonte desses programas encontra-se no Apêndice A.

- *Livermore Loop número 24* – encontra o índice do menor componente de um vetor de números inteiros [MCMA83];
- *Branch* – descrito em [DITZ87], conta os números pares e ímpares de um intervalo;

- *BCD-bin* – converte um número inteiro codificado em decimal, para o correspondente codificado em binário [AUHT85];
- *Simpson* – avalia o valor da integral de uma função pelo método de Simpson [CONT65]; e
- *Árvore* – gera uma árvore binária ordenada em um vetor [WIRT76].

A Tabela 5.2 apresenta algumas características estruturais desses programas. Ela mostra o número de blocos básicos de cada programa, o número médio de instruções por bloco básico e o número médio de ciclos de processador requerido por cada bloco básico dos programas de teste. Esses números de ciclos foram obtidos através da execução *seqüencial* dos programas da bateria, e eles refletem o tempo de latência das unidades funcionais do processador (instruções podem consumir de 1 a 4 ciclos de máquina, dependendo do seu tipo).

Programa Exemplo	Nº de Blocos Básicos	Nº Médio de Inst. / BB	Nº Médio de Ciclos / BB
Livermore	9	6,77	14,77
Branch	10	4,60	9,80
BCD	20	5,35	11,75
Simpson	24	7,25	16,70
Árvore	31	6,25	12,58

BB → Bloco Básico

Tabela 5.2: Características dos Programas Teste

Os valores da Tabela 5.2 podem ser assim interpretados: o programa *BCD* é formado por 20 blocos básicos, cada um contendo em média 5,35 instruções. Durante a execução seqüencial, cada bloco básico do programa *BCD* requer em média 11,75 ciclos de processador.

5.5 Avaliando a Qualidade do Código

Para cada programa de teste e para cada configuração do modelo CONDEX, geramos dois programas objeto: o código compactado localmente e o programa em linguagem de máquina compactado condicionalmente. Para efeito de comparação, utilizamos o

programa seqüencial como referência. Dessa forma, geramos sete programas objeto para cada componente da bateria de teste, conforme listado a seguir:

$$\text{programas de teste} \Rightarrow \left\{ \begin{array}{l} \text{seqüencial} \quad \{ \text{Máquina referência} \\ \\ \text{compactação local} \quad \left\{ \begin{array}{l} \text{Configuração 1} \\ \text{Configuração 2} \\ \text{Configuração 3} \end{array} \right. \\ \\ \text{compactação condicional} \quad \left\{ \begin{array}{l} \text{Configuração 1} \\ \text{Configuração 2} \\ \text{Configuração 3} \end{array} \right. \end{array} \right.$$

Para cada programa da bateria, contabilizamos o número de instruções, o número de IMFs e o número de ciclos de processador requerido por cada uma das sete versões de código objeto.

5.5.1 O Programa “Livermore Loop”

O código seqüencial do programa *Livermore Loop* é formado por 62 instruções distribuídas em 9 blocos básicos. A Figura 5.1 mostra o grafo do fluxo de controle (GFC) desse programa. Da mesma forma como foi apresentado nos três exemplos do capítulo anterior, cada nó do GFC representa um bloco básico, e o seu interior indica as instruções do bloco. Na parte externa do nó temos a freqüência de execução de cada bloco básico. Por exemplo, o terceiro bloco básico é composto pelas instruções 29, 30, 31 e 32 e foi executado 701 vezes (vide Figura 5.1).

Observando ainda a Figura 5.1, vemos que três blocos básicos (um composto pela instrução 50, um outro pelos comandos 51 e 52, e o terceiro pelas instruções 53, ... , 57) em conjunto com a instrução de um quarto bloco (a instrução 49), estão no interior da região tracejada. Somente essas instruções foram submetidas à compactação condicional em nossos experimentos. Nesse caso, realizamos a compactação condicional de um par do tipo “bt e bs”. As instruções de desvio incondicional 49 e 50 foram geradas pelo compilador para garantir a execução do bloco “THEN” ou do bloco “ELSE,” de acordo com o resultado da condição do comando “IF.” Durante a compactação condicional, essas instruções geralmente são eliminadas.

A Tabela 5.3 lista o número de instruções e o número de IMFs das sete formas executáveis do programa *Livermore Loop*.

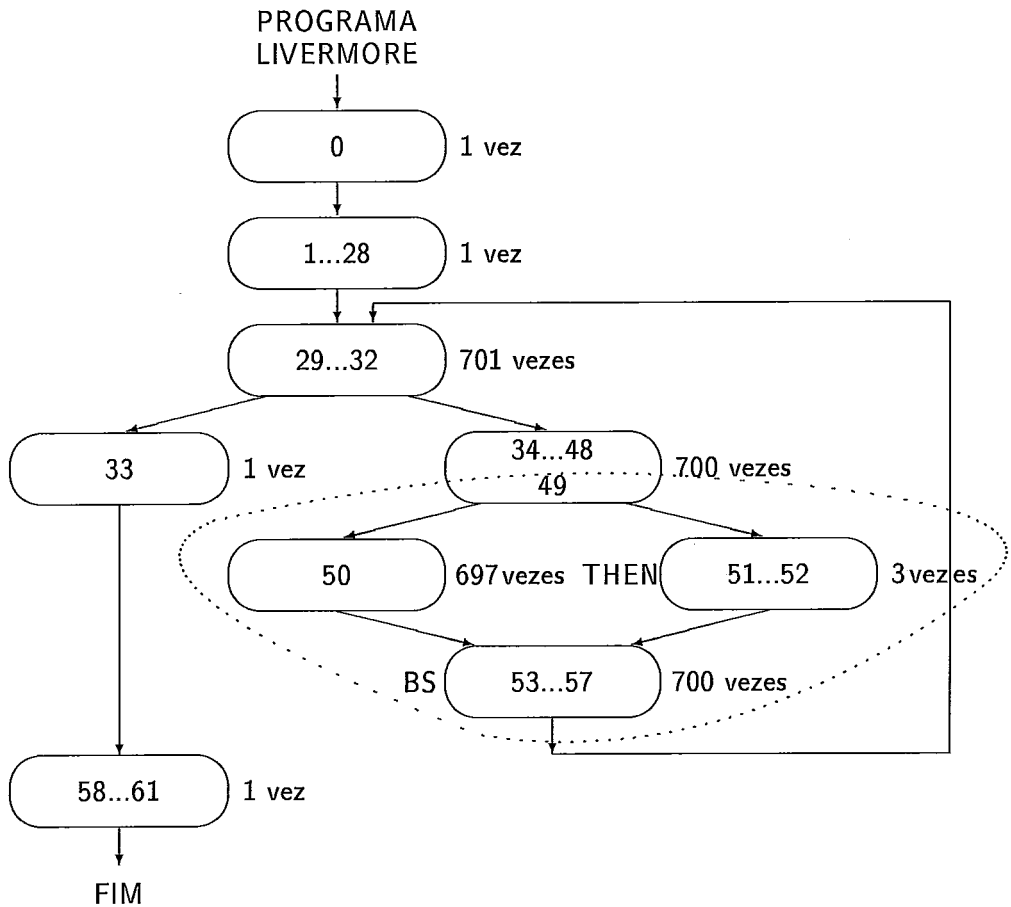


Figura 5.1: O Grafo de Fluxo de Controle do Programa *Livermore Loop*

Código		Nº de Instruções	Nº de IMFs
SEQUENCIAL		62	133
LOCAL	Configuração 1	62	112
	Configuração 2	62	76
	Configuração 3	62	64
COND.	Configuração 1	60	110
	Configuração 2	60	66
	Configuração 3	60	54

Tabela 5.3: Efeito da Compactação Local e Condicional no Programa *Livermore Loop*

Na Tabela 5.3 podemos observar que ocorreu uma redução no tamanho do programa executável para cada uma das configurações propostas. O programa

seqüencial é formado por 62 instruções distribuídas em 133 IMFs (o tempo de latência das operações provocou a introdução de NOPs, e por esse motivo, foram requeridas 133 IMFs para acomodar as 62 instruções do programa seqüencial). No código resultante da compactação local para a Configuração 1, temos que as 133 IMFs seqüenciais foram reduzidas a 112 instruções longas. Já na Configuração 2, o número de IMFs passou para 76, caíndo finalmente para 64 na Configuração 3.

Na Tabela 5.3 podemos ver que para a Configuração 1, o código obtido por meio da compactação condicional, as 112 IMFs compactadas localmente foram reduzidas a 110 instruções longas. Na Configuração 2, o número de IMFs passou para 66, chegando a 54 na Configuração 3. A tabela mostra ainda que o algoritmo de compactação local não modificou o número de instruções, porém o de compactação condicional reduziu esse número para 60. Tal constatação pode parecer estranha, já que a equivalência semântica de cada programa deve ser preservada quando da sua paralelização. Essa redução resulta da eliminação de duas instruções de desvio que tornaram-se desnecessárias quando as IMFs do par “bt e bs,” são compactadas.

É importante lembrar que embora seja possível eliminar algumas instruções de desvio durante o processo de compactação condicional, há casos em que instruções de desvio são acrescentadas. Isso ocorre quando os tamanhos dos blocos “bt e be” ou “bt e bs” forem muito diferentes, fazendo com que ao final do bloco com menor número de instruções, seja adicionada uma instrução de desvio para saltar as instruções do bloco maior, evitando-se desse modo o consumo de ciclos de máquina desnecessários durante a execução. Em outras palavras, a inclusão de uma instrução de desvio evita, em tempo de execução, a busca de instruções longas que não contêm nenhuma instrução pertencente ao bloco menor.

O efeito das compactações local e condicional durante a interpretação do programa *Livermore Loop número 24*, é mostrado na Tabela 5.4. Essa tabela apresenta o número de instruções e de IMFs executadas durante interpretação do programa teste em cada uma das configurações do modelo e na máquina seqüencial. A terceira coluna da tabela lista a taxa de temos a aceleração (*speedup*) obtida na execução em relação a execução seqüencial.

O total de ciclos de processador (i.e., o número de IMFs executadas) consumidos durante a interpretação do programa na Configuração 1 é praticamente o mesmo requerido pela execução seqüencial (*speedup* igual a 1,001).

Código		Inst. Executadas	IMFs Executadas	Speedup
SEQUENCIAL		18241	34404	1,000
LOCAL	Configuração 1	18241	30188	1,140
	Configuração 2	18241	21764	1,581
	Configuração 3	18241	21752	1,582
COND.	Configuração 1	16844	34376	1,001
	Configuração 2	16844	20343	1,691
	Configuração 3	16844	20331	1,692

Tabela 5.4: Efeito da Compactação Local e Condicional na Execução do Programa *Livermore Loop*

A Configuração 1 inclui poucas unidades funcionais. Por esse motivo, o algoritmo não foi capaz de movimentar um número significativo de instruções do bloco “bs” para as IMFs do bloco “bt.” Conseqüentemente, as IMFs do bloco “bs” tornaram-se mais vazias, mas não puderam ser eliminadas.

Em tempo de execução, esse baixo índice de compactação penaliza o desempenho do processador: um maior número de IMFs são buscadas quando a condição do “IF” for falsa (melhor seria se as instruções do bloco “bs” não estivessem tão espalhadas). Quando a condição for verdadeira, o tempo de execução não é penalizado. Se dispusermos *a priori* do perfil de execução do programa de teste, poderemos evitar essa queda no desempenho.

O problema não repetiu-se nas outras configurações. O paralelismo potencial provido por um número maior de unidades funcionais garantiu uma taxa mais elevada de compactação, aumentando o *speedup* dos dois processadores.

A queda no desempenho sugere a necessidade de se estabelecer critérios para a compactação das instruções de um bloco “bs” nas IMFs do bloco “THEN.” Esses critérios poderiam levar em conta o número de instruções de cada bloco e a quantidade de unidades funcionais da arquitetura.

O histograma na Figura 5.2, apresenta os tempos de execução do programa compactado localmente e condicionalmente nas três configurações. Esses tempos, expressos em percentagens, são representados em termos do tempo de execução

requerido pela máquina seqüencial.

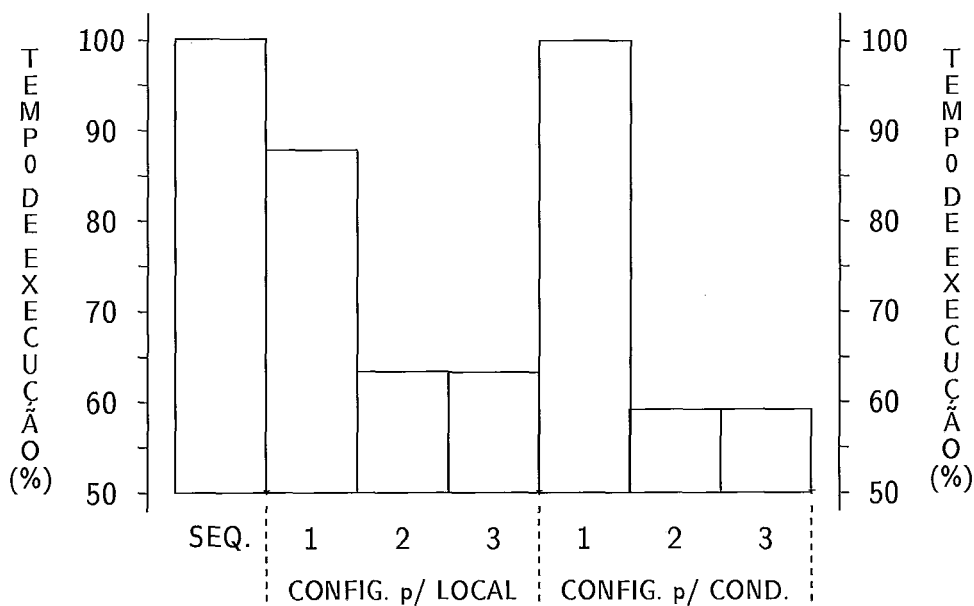


Figura 5.2: Execução Seqüencial \times Execuções com Compactação Local e Condicional do Programa *Livermore*

5.5.2 O Programa “Branch”

Na Figura 5.3 vemos o GFC do programa *Branch*. O código executável seqüencial compreende um total de 47 instruções que formam seus 10 blocos básicos. Do mesmo modo que no programa *Livermore Loop número 24*, no GFC do programa *Branch* cada nó representa um bloco básico. No interior dos blocos básicos existe a indicação de quais instruções fazem parte de cada um deles, e na parte externa a informação de quantas vezes cada bloco básico foi executado durante a interpretação do programa quando da realização dos nossos testes.

Na Figura 5.3 demarcamos os blocos básicos que foram submetidos à compactação condicional, por meio da curva tracejada. Nesse programa, como podemos ver na figura, os três blocos básicos (um composto pela instrução 28, um outro pelas instruções 29, ... , 33, e um último formado pelas instruções 34, ... , 37), e uma instrução de um quarto bloco (a instrução 27) forneceram as instruções para a compactação condicional. Os blocos submetidos à compactação condicional, corres-

pondem ao par “bt e be” como ilustrado na figura.

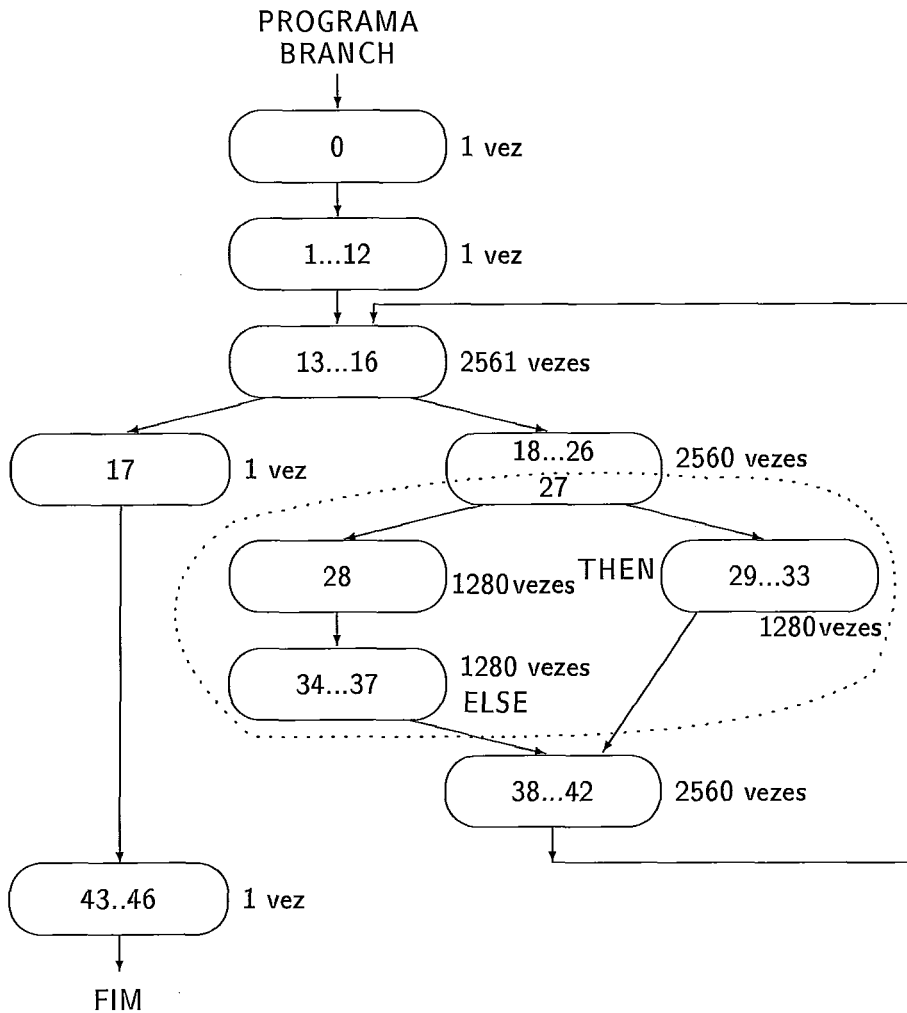


Figura 5.3: O Grafo de Fluxo de Controle do Programa *Branch*

Observando a Tabela 5.5, vemos que as 47 instruções do código seqüencial ocupam 98 IMFs. Depois de realizada a compactação local o código executável em cada uma das configurações do simulador do CONDEX, apresenta as mesmas 47 instruções. Após a compactação condicional esse número é de apenas 44 instruções. Nesse caso três instruções de desvio foram eliminadas pelo processo de compactação condicional que fez a fusão das instruções do par “bt e be”: a instrução 27 que faz o desvio do fluxo de controle para o “bt” quando a condição testada for verdadeira, instrução 28 que desvia o fluxo de controle para o “be” quando a condição testada for falsa, e a instrução 33, que é a última instrução do “bt” e que salta as instruções do “be” desviando o controle para “bs.”

O número de IMFs dos programas executáveis compactados localmente é de 85, 68 e 63 respectivamente, para a Configuração 1, 2 e 3. Para essas configurações, o programa submetido à compactação condicional, apresentam 73, 56 e 51 instruções (Tabela 5.5).

Código		Nº de Instruções	Nº de IMFs
SEQUENCIAL		47	98
LOCAL	Configuração 1	47	85
	Configuração 2	47	68
	Configuração 3	47	63
COND.	Configuração 1	44	73
	Configuração 2	44	56
	Configuração 3	44	51

Tabela 5.5: Efeito da Compactação Local e Condicional no Programa *Branch*

Código		Inst. Executadas	IMFs Executadas	Speedup
SEQUENCIAL		61462	130603	1,000
LOCAL	Configuração 1	61462	117795	1,109
	Configuração 2	61462	94747	1,378
	Configuração 3	61462	92183	1,416
COND.	Configuração 1	56342	112675	1,159
	Configuração 2	56342	89627	1,457
	Configuração 3	56342	87063	1,500

Tabela 5.6: Efeito da Compactação Local e Condicional na Execução do Programa *Branch*

Vemos na Tabela 5.6 que para o programa *Branch*, o *speedup* obtido por meio da compactação condicional é sempre maior que o atingido através da compactação local. Na Configuração 1, a aceleração para a compactação condicional é de 1,159. Quando dobramos o número de cada tipo das unidades funcionais do processador (Configuração 2), vemos que a aceleração atinge 1,457. Isto significa que o paralelismo de baixo nível, existente no programa não foi totalmente explorado na Configuração 1, por causa da falta de recursos. Quando mais uma vez dobramos o número de unidades funcionais da Configuração 2 e para obter a Configuração 3, a aceleração passou de 1,457 para 1,500. Isso mostra que o paralelismo no nível de

instrução existente já havia sido quase completamente explorado pelo escalonamento feito para a Configuração 2.

O histograma na Figura 5.4, apresenta os tempos de execução do programa compactado localmente e condicionalmente nas três configurações. Esses tempos, expressos em percentagens, são representados em termos do tempo de execução requerido pela máquina seqüencial.

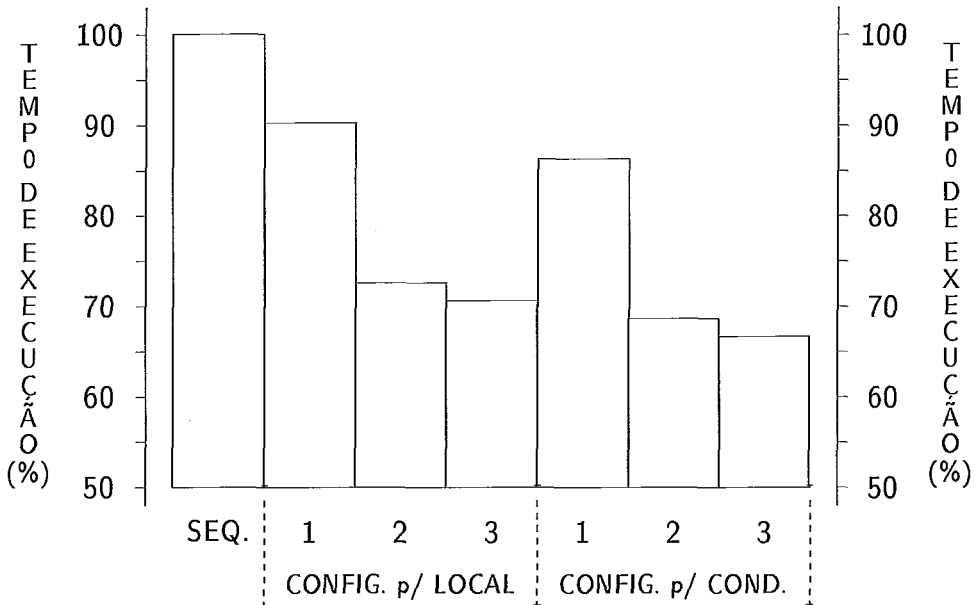


Figura 5.4: Execução Seqüencial × Execuções com Compactação Local e Condicional do Programa *Branch*

5.5.3 O Programa “BCD”

Como podemos ver na Figura 5.5 o GFC do programa *BCD* é composto de 20 blocos básicos nos quais estão as 107 instruções que formam o código executável seqüencial. Como nos dois programas anteriormente examinados, as instruções que fazem parte de cada bloco básico estão indicadas no interior de cada nó, e o número de vezes que cada bloco básico foi executado durante a realização dos testes, está indicado na parte externa de cada nó do grafo. Nesse programa as instruções submetidas à compactação condicional foram as pertencentes aos blocos básicos compostos pelas instruções 54; 55, ... , 58; e 59, ... , 71. A instrução 53 também foi fazer parte das

instruções compactadas condicionalmente. No caso do programa *BCD* compactamos um “bt” com um “bs.”

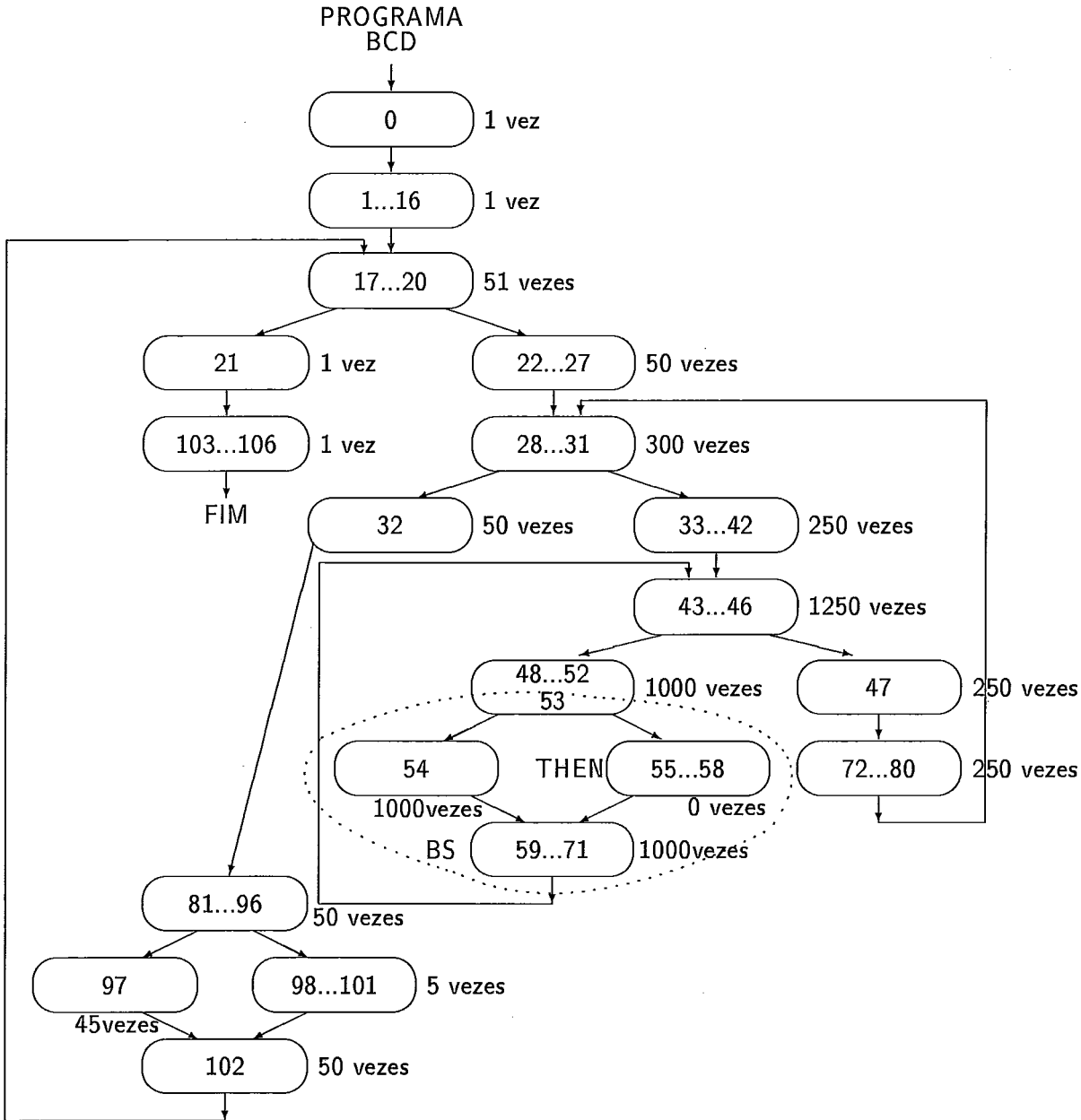


Figura 5.5: O Grafo de Fluxo de Controle do Programa *BCD*

O número de instruções e o número de IMFs de cada uma das formas executáveis obtidas para o programa *BCD*, são mostradas na Tabela 5.7. Para o código seqüencial temos 107 instruções ocupando 235 IMFs. Os códigos gerados por meio da compactação local para as Configurações 1, 2 e 3, possuem 107 instruções ocu-

pando respectivamente 203, 145 e 118 IMFs. Os programas executáveis obtidos através da compactação condicional apresentam 105 instruções em 201, 142 e 111 IMFs.

Código		Nº de Instruções	Nº de IMFs
SEQUENCIAL		107	235
LOCAL	Configuração 1	107	203
	Configuração 2	107	145
	Configuração 3	107	118
COND.	Configuração 1	105	201
	Configuração 2	105	142
	Configuração 3	105	111

Tabela 5.7: Efeito da Compactação Local e Condicional no Programa *BCD*

Resumimos o efeito das compactações local e condicional durante a interpretação do programa *BCD*, na Tabela 5.8.

Código		Inst. Executadas	IMFs Executadas	Speedup
SEQUENCIAL		33072	72772	1,000
LOCAL	Configuração 1	33072	63507	1,145
	Configuração 2	33072	45937	1,584
	Configuração 3	33072	36332	2,002
COND.	Configuração 1	31199	72983	0,997
	Configuração 2	31199	50921	1,429
	Configuração 3	31199	37315	1,950

Tabela 5.8: Efeito da Compactação Local e Condicional na Execução do Programa *BCD*

Na Tabela 5.8 observamos que a compactação local sempre produz tempos de execução menores que a condicional. Na Configuração 1 o tempo de execução do programa *BCD* aumenta até mesmo com relação a execução seqüencial, quando a compactação condicional é usada, isto é, o *speedup* é de 0,997. Essa anomalia ocorreu pois nesse caso específico foram compactadas instruções de um “bs” com instruções de um “bt.” Como na Configuração 1, o número de unidades funcionais presentes é reduzido, o número de instruções movidas do “bs” para as instruções

longas do “bt” foi pequeno, não reduzindo o número de IMF’s daquele (as IMF’s do “bs” apenas ficaram mais vazias mas não puderam ser eliminadas), resultando num maior número de instruções longas buscadas quando a condição que faz com que as instruções do “bt” sejam executadas, é falsa. Quando a condição que faz com que as instruções do “bt” sejam executadas é verdadeira, o número de IMF’s buscadas não aumenta. Contudo, durante a execução do programa teste o “bt” não é executado (i.e., é executado 0 vezes como mostra o GFC da Figura 5.5). Nas outras duas configurações, a situação se repete porém com menor gravidade. O problema nas Configurações 2 e 3 da arquitetura, é reduzido pela introdução de um número maior de unidades funcionais na arquitetura.

Diante da situação descrita, somos levados a pensar que será preciso estabelecer algum critério para a compactação das instruções de blocos sucessores em IMF’S que contém instruções de blocos THEN. Esse critério deverá levar em conta o número de instruções de cada bloco e a quantidade de recursos disponíveis na arquitetura.

Uma outra forma de observar os resultados da Tabela 5.8 é através do histograma mostrado na Figura 5.6.

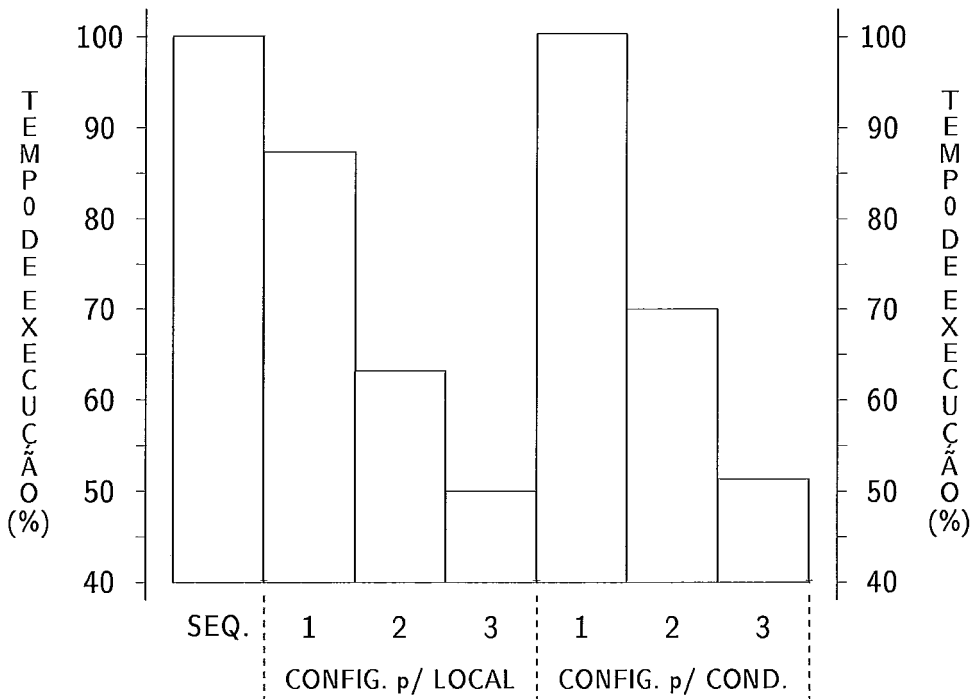


Figura 5.6: Execução Sequencial × Execuções com Compactação Local e Condicional do Programa *BCD*

5.5.4 O Programa “Simpson”

Na Figura 5.7 vemos o grafo de fluxo de controle do programa *Simpson*. Esse programa possui dois procedimentos (*Simp* e *Fun*), que aparecem nas Figuras 5.8 e 5.9. O procedimento *Simp* é chamado pelo programa principal, que por sua vez chama o procedimento *Fun* como é possível observar nas Figuras 5.7, 5.8 e 5.9.

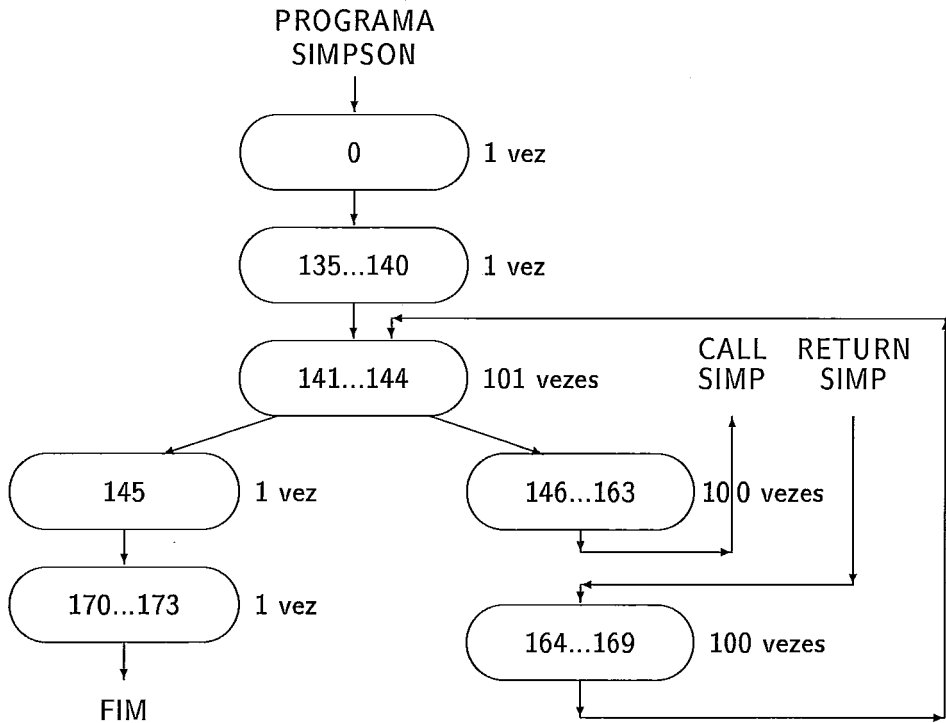


Figura 5.7: O Grafo de Fluxo de Controle do Programa *Simpson*

Os trechos do programa *Simpson*, submetidos à compactação condicional fazem parte do procedimento *Simp* que aparece na Figura 5.8. Nesse caso as duas curvas tracejadas delimitam os dois trechos do código que tiveram suas instruções compactadas condicionalmente. Trata-se em ambos os casos de pares blocos “bt e be”.

Como podemos observar nas Figuras 5.7, 5.8 e 5.9, do mesmo modo que em todos os outros grafos de fluxo de controle dos programas examinados anteriormente, também para o programa *Simpson*, mostramos em cada bloco básico as instruções presentes em cada bloco básico e o número de vezes que cada um deles foi executado durante nossos testes.

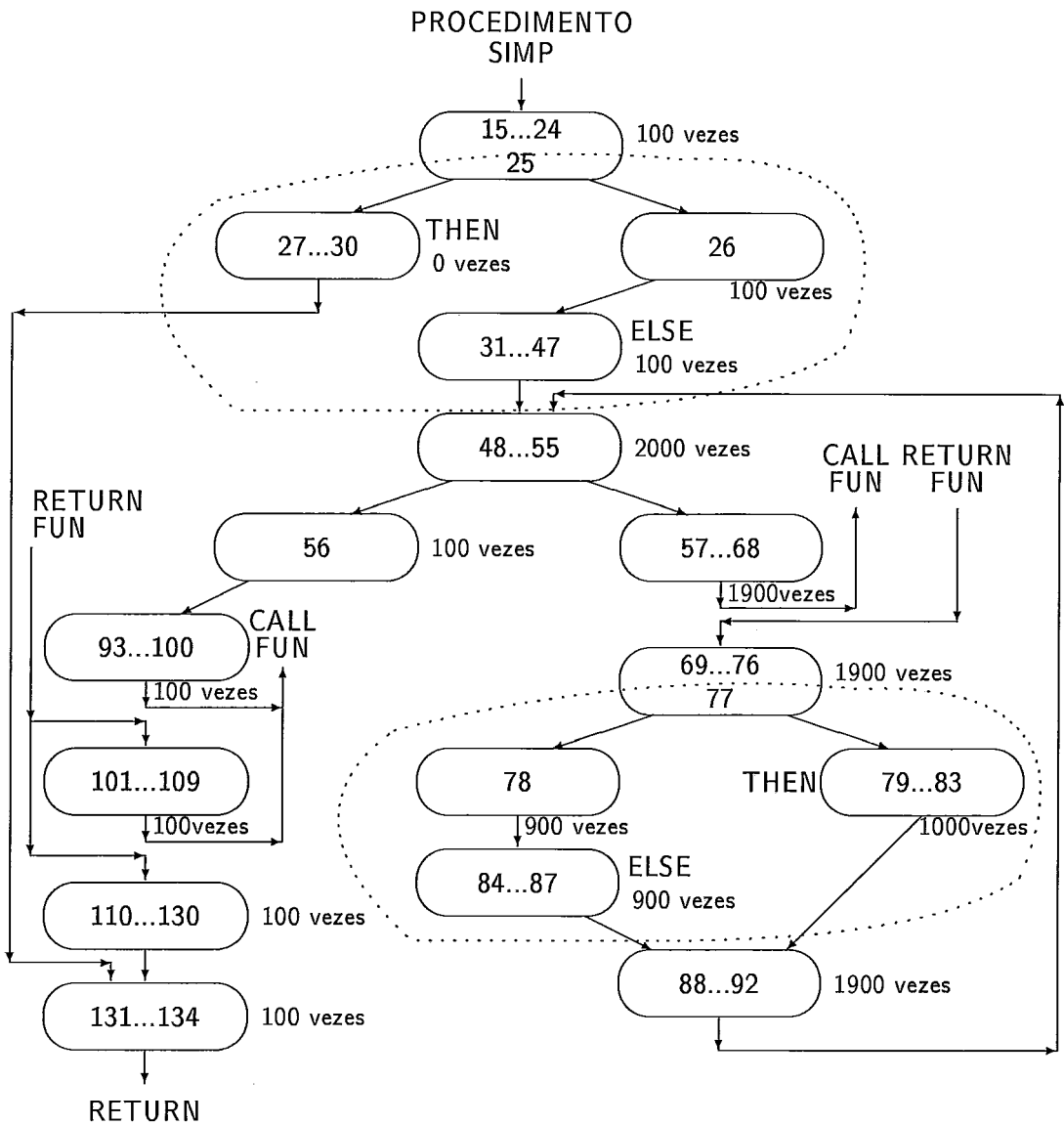


Figura 5.8: O Grafo de Fluxo de Controle do Procedimento *Simp* do Programa *Simpson*

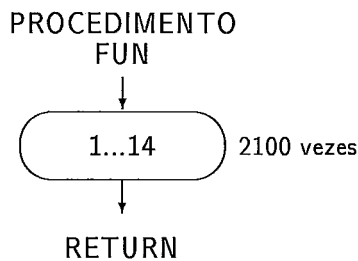


Figura 5.9: O Grafo de Fluxo de Controle do Procedimento *Fun* do Programa *Simpson*

Observando a Tabela 5.9, vemos que depois de realizada a compactação local, o código executável apresenta as mesmas 174 instruções do código seqüencial, em todas as configurações. Para a compactação condicional, o número de instruções na Configuração 1 é 170 e 169 para as outras duas configurações do simulador CONDEX. Essa diferença no número de instruções foi causada pela inclusão de uma instrução de desvio no código gerado para Configuração 1. A presença dessa instrução evita que IMFs que contêm apenas instruções do “be” continuem sendo buscadas durante a execução do “bt”. Nas outras duas configurações isso não foi necessário pois o maior número de recursos, permitiu um casamento mais harmonioso dos dois blocos compactados com código de condição. Os códigos executáveis obtidos por meio da compactação local, para as três configurações, apresentam respectivamente 334, 233 e 212 IMFs. Já os obtidos por meio da compactação condicional, contêm 308, 210 e 188 IMFs.

Código		Nº de Instruções	Nº de IMFs
SEQUENCIAL		174	401
LOCAL	Configuração 1	174	334
	Configuração 2	174	233
	Configuração 3	174	212
COND.	Configuração 1	170	308
	Configuração 2	169	210
	Configuração 3	169	188

Tabela 5.9: Efeito da Compactação Local e Condicional no Programa *Simpson*

Mostramos o efeito da compactação local e da compactação condicional durante a interpretação do programa *Simpson* na Tabela 5.10.

Examinando a Tabela 5.10, vemos que nas três configurações utilizadas nos nossos experimentos, o número de ciclos de processador consumidos para a interpretação dos programas compactados condicionalmente é sempre menor que o dos obtidos através da compactação local. Para a execução seqüencial 269828 IMFs foram executadas. Já a execução dos programas compactados localmente consumiu 227923, 163720 e 153620 ciclos de processador. Durante a execução dos códigos compactados com códigos de condição foram executadas 225723, 161620 e 149620 IMFs nas Configurações 1, 2 e 3 respectivamente.

Código		Inst. Executadas	IMFs Executadas	Speedup
SEQUENCIAL		114316	269828	1,000
LOCAL	Configuração 1	114316	227923	1,183
	Configuração 2	114316	163720	1,648
	Configuração 3	114316	153620	1,756
COND.	Configuração 1	111316	225723	1,195
	Configuração 2	110316	161620	1,669
	Configuração 3	110316	149620	1,803

Tabela 5.10: Efeito da Compactação Local e Condicional na Execução do Programa *Simpson*

O histograma na Figura 5.10, apresenta os tempos de execução do programa compactado localmente e condicionalmente nas três configurações. Esses tempos, expressos em percentagens, são representados em termos do tempo de execução requerido pela máquina seqüencial.

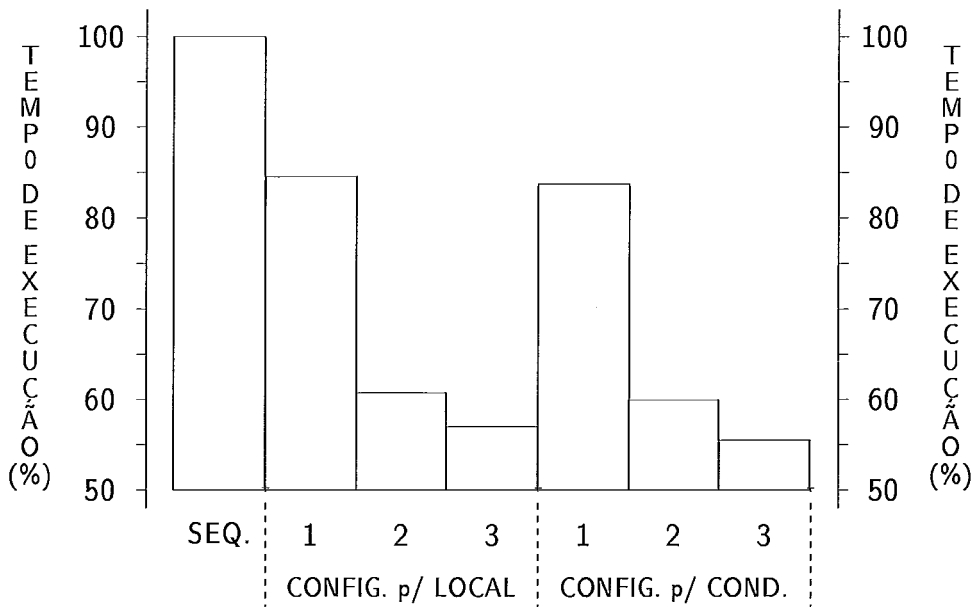


Figura 5.10: Execução Seqüencial × Execuções com Compactação Local e Condicional do Programa *Simpson*

5.5.5 O Programa “Árvore”

Finalmente temos o programa *Árvore* que é formado pelo programa principal e por um procedimento (*Inserere*), cujos grafos de fluxo de controle são mostrados nas Figuras 5.11 e 5.12, respectivamente.

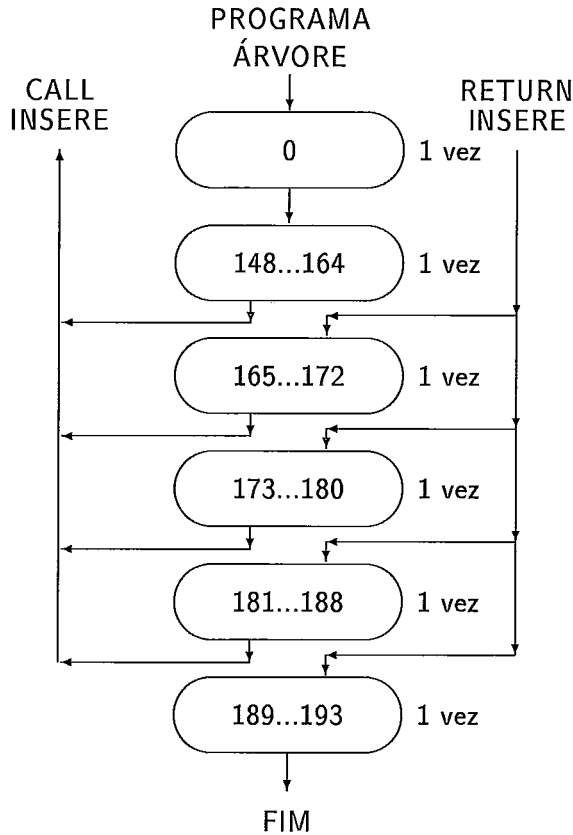


Figura 5.11: O Grafo de Fluxo de Controle do Programa *Árvore*

Podemos observar nas Figuras 5.11 e 5.12 do mesmo modo que em todos os outros grafos de fluxo de controle dos programas examinados anteriormente, também para esse programa mostramos em cada bloco básico as instruções presentes em cada um deles e o número de vezes que cada um deles foi executado durante execução seqüencial.

O programa *Árvore* possui ao todo 31 blocos básicos representados por meio dos nós dos grafos de fluxo de controle das Figuras 5.11 e 5.12. Durante a execução o programa chama o procedimento *Inserere*. No procedimento *Inserere* que aparece na

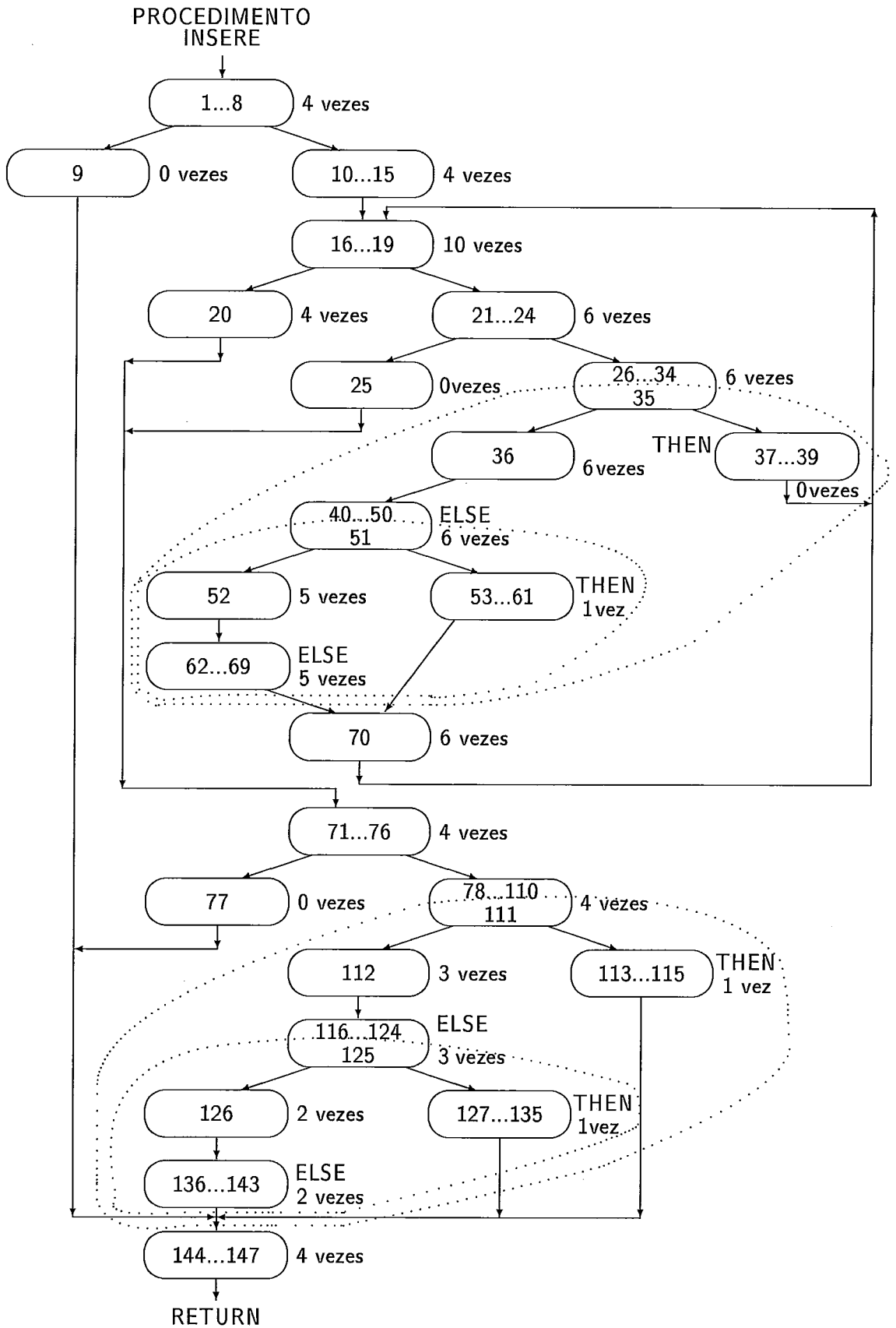


Figura 5.12: O Grafo de Fluxo de Controle do Procedimento *Inserer* do Programa *Árvore*

Figura 5.12 temos os quatro trechos distintos submetidos à compactação condicional. Na figura vemos quatro curvas tracejadas delimitando os quatro trechos do código que tiveram suas operações compactadas condicionalmente. Trata-se em todos os casos de pares de “bt e be”.

O código executável seqüencial do programa *Árvore*, compreende um total de 194 instruções contidas em 390 IMFs. Seis dos blocos básicos fazem parte do programa principal e os 25 restantes do procedimento *Inserere*.

O número de instruções e o número de IMFs de cada uma das formas executáveis obtidas para o programa *Árvore*, são mostrados na Tabela 5.11. Os programas executáveis obtidos por meio da compactação local possuem 194 instruções e que ocupam nas três configurações 324, 249 e 224 IMFs. Já os códigos gerados por meio da compactação condicional, para executar nas Configurações 1, 2 e 3, possuem 184 instruções ocupando respectivamente 271, 197 e 171 IMFs.

Código		Nº de Instruções	Nº de IMFs
SEQUENCIAL		194	390
LOCAL	Configuração 1	194	324
	Configuração 2	194	249
	Configuração 3	194	224
COND.	Configuração 1	184	271
	Configuração 2	184	197
	Configuração 3	184	171

Tabela 5.11: Efeito da Compactação Local e Condicional no Programa *Árvore*

O efeito das compactações local e condicional durante a interpretação do programa *Árvore*, aparece nos resultados contidos na Tabela 5.12.

Na Tabela 5.12 vemos que o número de IMFs executadas é sempre menor no caso da compactação condicional. Na execução seqüencial, 1241 ciclos de máquina são consumidos, já nas Configurações 1, 2 e 3, durante a execução condicional, esse número é reduzido para 1025, 760 e 669 respectivamente.

O histograma na Figura 5.13, apresenta os tempos de execução do programa compactado localmente e condicionalmente nas três configurações. Esses tempos, expressos em percentagens, são representados em termos do tempo de execução requerido pela máquina seqüencial. Observamos no histograma que o tempo gasto

na sua execução quando submetido a compactação condicional, foi reduzido nas três configurações usadas.

Código		Inst. Executadas	IMFs Executadas	Speedup
SEQUENCIAL		608	1241	1,000
LOCAL	Configuração 1	608	1063	1,167
	Configuração 2	608	795	1,561
	Configuração 3	608	707	1,755
COND.	Configuração 1	571	1025	1,211
	Configuração 2	571	760	1,633
	Configuração 3	571	669	1,855

Tabela 5.12: Efeito da Compactação Local e Condicional na Execução do Programa *Árvore*

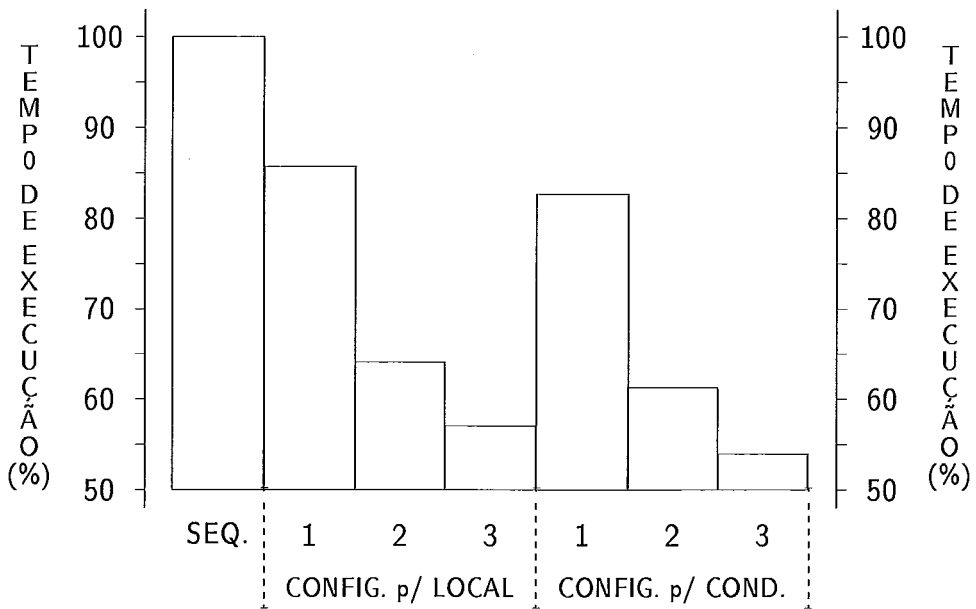


Figura 5.13: Execução Sequencial × Execuções com Compactação Local e Condicional do Programa *Árvore*

Na Seção 5.6 apresentamos diversas tabelas contendo o resumo dos valores obtidos durante nossos experimentos.

5.6 Sumário dos Resultados Obtidos

Para possibilitar ao leitor uma visão global dos resultados obtidos, apresentamos nessa seção, 14 tabelas que refletem o efeito da compactação local e da condicional em cada programa de teste e a sua influência no tempo de execução desses.

Programa	Nº de IMFs	Nº de Instruções
Livermore	133	62
Branch	98	47
BCD	235	107
Simpson	401	174
Árvore	390	194

Tabela 5.13: N^{os} de IMFs e de Instruções Obtidos para a Execução Seqüencial

Inicialmente na Tabela 5.13 relacionamos o número de IMFs e de instruções de cada programa de teste, do código executável seqüencial. Desse modo torna-se possível estabelecer paralelo com os valores obtidos quando da aplicação das compactações local e condicional.

Nas Tabelas 5.14, 5.15, 5.16, 5.17, 5.18 e 5.19 apresentamos o número de IMFs e o número de instruções dos códigos executáveis de cada programa de teste submetido à compactação local e condicional, para cada uma das configurações do modelo CONDEX.

Programa	Nº de IMFs	Nº de Instruções
Livermore	112	62
Branch	85	47
BCD	203	107
Simpson	334	174
Árvore	324	194

Tabela 5.14: N^{os} de IMFs e de Instruções Obtidos por meio da Compactação Local para a Configuração 1

Programa	Nº de IMFs	Nº de Instruções
Livermore	110	60
Branch	73	44
BCD	201	105
Simpson	308	170
Árvore	271	184

Tabela 5.15: N^{os} de IMFs e de Instruções Obtidos por meio da Compactação Condicional para a Configuração 1

Programa	Nº de IMFs	Nº de Instruções
Livermore	76	62
Branch	68	47
BCD	145	107
Simpson	233	174
Árvore	249	194

Tabela 5.16: N^{os} de IMFs e de Instruções Obtidos por meio da Compactação Local para a Configuração 2

Programa	Nº de IMFs	Nº de Instruções
Livermore	66	60
Branch	56	44
BCD	142	105
Simpson	210	169
Árvore	197	184

Tabela 5.17: N^{os} de IMFs e de Instruções Obtidos por meio da Compactação Condicional para a Configuração 2

Programa	Nº de IMFs	Nº de Instruções
Livermore	64	62
Branch	63	47
BCD	118	107
Simpson	212	174
Árvore	224	194

Tabela 5.18: N^{os} de IMFs e de Instruções Obtidos por meio da Compactação Local para a Configuração 3

Programa	Nº de IMFs	Nº de Instruções
Livermore	54	60
Branch	51	44
BCD	111	105
Simpson	188	169
Árvore	171	184

Tabela 5.19: N^{os} de IMFs e de Instruções Obtidos por meio da Compactação Condicional para a Configuração 3

Na Tabela 5.20 estão registrados o número de IMFs e de instruções executadas do código seqüencial, para cada programa do *benchmark* empregado.

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas
Livermore	34404	18241
Branch	130603	61462
BCD	72772	33072
Simpson	269828	114316
Árvore	1241	608

Tabela 5.20: N^{os} de IMFs e de Instruções Executadas Seqüencialmente

Os resultados dinâmicos da compactação local e condicional, isto é, o efeito da compactação condicional durante a interpretação dos programas teste, são apresentados nas Tabelas 5.21, 5.22, 5.23, 5.24, 5.25 e 5.26. Essas tabelas além dos números de IMFs e instruções executadas, incluem o *speedups* obtidos para cada programa e o *speedup* médio observado em cada configuração.

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas	Speedup
Livermore	30188	18241	1,140
Branch	117795	61462	1,109
BCD	63507	33072	1,145
Simpson	227923	114316	1,183
Árvore	1063	608	1,167

Speedup médio: 1,148

Tabela 5.21: Nºs de IMFs e Instruções Compactadas Localmente Executadas na Configuração 1

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas	Speedup
Livermore	34376	16844	1,001
Branch	112675	56342	1,159
BCD	72983	31199	0,997
Simpson	225723	111316	1,195
Árvore	1025	571	1,211

Speedup médio: 1,113

Tabela 5.22: Nºs de IMFs e Instruções Compactadas Condicionalmente Executadas na Configuração 1

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas	Speedup
Livermore	21764	18241	1,581
Branch	94747	61462	1,378
BCD	45937	33072	1,584
Simpson	163720	114316	1,648
Árvore	795	608	1,561

Speedup médio: 1,550

Tabela 5.23: Nºs de IMFs e Instruções Compactadas Localmente Executadas na Configuração 2

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas	Speedup
Livermore	20343	16844	1,691
Branch	89627	56342	1,457
BCD	50921	31199	1,429
Simpson	161620	110316	1,669
Árvore	760	571	1,633

Speedup médio: 1,576

Tabela 5.24: N^{os} de IMFs e Instruções Compactadas Condicionalmente Executadas na Configuração 2

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas	Speedup
Livermore	21752	18241	1,582
Branch	92183	61462	1,416
BCD	36331	33072	2,002
Simpson	153620	114316	1,756
Árvore	707	608	1,755

Speedup médio: 1,702

Tabela 5.25: N^{os} de IMFs e Instruções Compactadas Localmente Executadas na Configuração 3

Programa	Nº de IMFs Executadas	Nº de Instr. Executadas	Speedup
Livermore	20331	16844	1,692
Branch	87063	56342	1,500
BCD	37315	31199	1,950
Simpson	149620	110316	1,803
Árvore	669	571	1,855

Speedup médio: 1,760

Tabela 5.26: N^{os} de IMFs e Instruções Compactadas Condicionalmente Executadas na Configuração 3

Além dos valores observados durante a interpretação dos programas de teste que foram apresentados nesse capítulo, mostramos no Apêndice B, o nível de ocupação das unidades funcionais em cada configuração derivada do modelo CONDEX. Na prática o potencial de processamento oferecido pela inclusão de múltiplas unidades

funcionais na arquitetura nem sempre é acompanhado pelo aumento proporcional no desempenho.

Capítulo 6

Técnicas de Escalonamento mais Agressivas

6.1 Introdução

Visando encontrar métodos alternativos para explorar mais eficientemente o paralelismo potencial do modelo CONDEX, decidimos realizar um conjunto de experimentos adicionais. Durante esses experimentos, utilizamos a mesma bateria de programas de teste, e geramos (manualmente) código executável empregando duas novas modalidades de algoritmo de compactação global, baseadas no perfil de execução dos programas de teste. Posteriormente interpretamos cada programa, nas três configurações da arquitetura especificadas no Capítulo 5.

Nesse capítulo, descrevemos as características das duas modalidades de compactação propostas, na Seção 6.2 e na Seção 6.3 apresentamos resultados obtidos pela compactação do código seqüencial e durante interpretação dos programas de testes. Finalmente na Seção 6.4 temos o resumo dos valores verificados durante os nossos experimentos.

6.2 Técnicas de Compactação Baseadas no Perfil de Execução dos Programas

As duas variações do algoritmo de escalonamento apresentadas, fazem uso da capacidade de execução condicional do nosso modelo de arquitetura.

Um algoritmo de compactação global baseado no perfil de execução dos programas e o *trace scheduling*. Essa técnica, desenvolvida em 1981 por Fisher [FISH81], destinava-se inicialmente à compactação global de μ -código. Posteriormente, a técnica foi adaptada e incorporada ao compilador Bulldog [ELLI86] (desenvolvido na Universidade de Yale), que gera código para a arquitetura VLIW, ELI-512 [FISH83].

Conforme apresentado no Capítulo 2, o *trace scheduling* é uma técnica de compactação que faz o escalonamento das instruções ao longo do fluxo de controle do programa como um todo. Isso significa que a técnica explora o paralelismo em trechos (*traces*) de execução do programa. A escolha dos trechos do programa compactados prioritariamente, baseia-se na probabilidade com que estes são executados. A probabilidade de execução de cada trecho do programa, pode ser fornecida pelo programador, ou determinada automaticamente. A movimentação de instruções ao longo dos *traces*, obriga a inclusão de código de reparo na última fase do processo de compactação, (i.e., código para anular os efeitos da movimentação de instruções para outros blocos básicos).

Chamamos de *Técnica baseada no Perfil de Execução* (ou TPE) ao processo de escalonamento de instruções em IMFs pertencentes a um trecho de programa previamente selecionado. O trecho inclui os blocos básicos mais freqüentemente executados. Em nossos experimentos, a freqüência de execução de cada trecho dos programas, foi determinada durante da realização dos testes da compactação condicional apresentados no Capítulo 5.

Quando da aplicação do TPE, cada bloco básico do programa é submetido à compactação local separadamente. Em seguida, os trechos do programa, mais freqüentemente executados são selecionados e segundo a técnica *list scheduling*, instruções de todo o trecho são movidas além das fronteiras dos blocos básicos respeitando as dependências de dados e restrições no uso de recursos.

Utilizamos também uma segunda modalidade de algoritmo de escalonamento de instruções baseada no perfil de execução dos programas. Nesse caso, o código previamente submetido à compactação condicional foi usado como entrada pelo processo de compactação. Desse modo, trechos de código com pares de blocos “bt e be” e “bt e bs” já escalonados por meio da compactação condicional, são submetidos ao nosso processo de escalonamento que usa como base o *trace scheduling*. Denominamos esse método de *Técnica baseada no Perfil de Execução +*, ou abreviadamente,

TPE+. Uma vez selecionado o trecho, o processo de escalonamento de instruções em IMFs, é o mesmo que o usado no TPE.

6.3 Os Experimentos

Nesses experimentos usamos os programas de teste e as configurações do modelo CONDEX, descritos no Capítulo 5.

Inicialmente, os trechos de programa executados mais freqüentemente foram identificados e compactados condicionalmente segundo as técnicas TPE e TPE+. Como resultado obtivemos as formas interpretáveis nas três configurações do modelo CONDEX.

Interpretando cada programa compactado nas três configurações da arquitetura, obtivemos então o número de instruções e executadas e o número de ciclos de processador consumidos pela interpretação.

Nas Subseções 6.3.1, 6.3.2, 6.3.3, 6.3.4 e 6.3.5, mostramos os valores obtidos durante os experimentos para cada programa de teste.

6.3.1 O Programa “Livermore Loop”

A Figura 6.1 apresenta o GFC do programa *Livermore Loop número 24*. No GFC cada nó representa um bloco básico. O interior de cada nó possui a indicação das instruções que fazem parte de cada um deles. Na parte externa de cada bloco, temos o número de vezes que esse foi executado durante a interpretação do código seqüencial, durante a realização dos nossos experimentos. Os blocos básicos que fazem parte do *trace* utilizado em nossos experimentos, aparecem no interior da figura tracejada. No caso do programa *Livermore Loop* o *trace* selecionado e submetido ao TPE, é formado por seis blocos básicos.

Na Figura 6.1 vemos que o *trace* selecionado não inclui os dois blocos usados na compactação condicional, isto é, o par “bt” “bs.” Nesse caso apenas o bloco “bs” faz parte do trecho submetido ao TPE.

No caso do TPE+, o trecho selecionado é semelhante ao usado no TPE. A diferença reside no fato de que no TPE+, os blocos “bt” e “bs” já submetidos à

compactação condicional, formam um único bloco no *trace*.

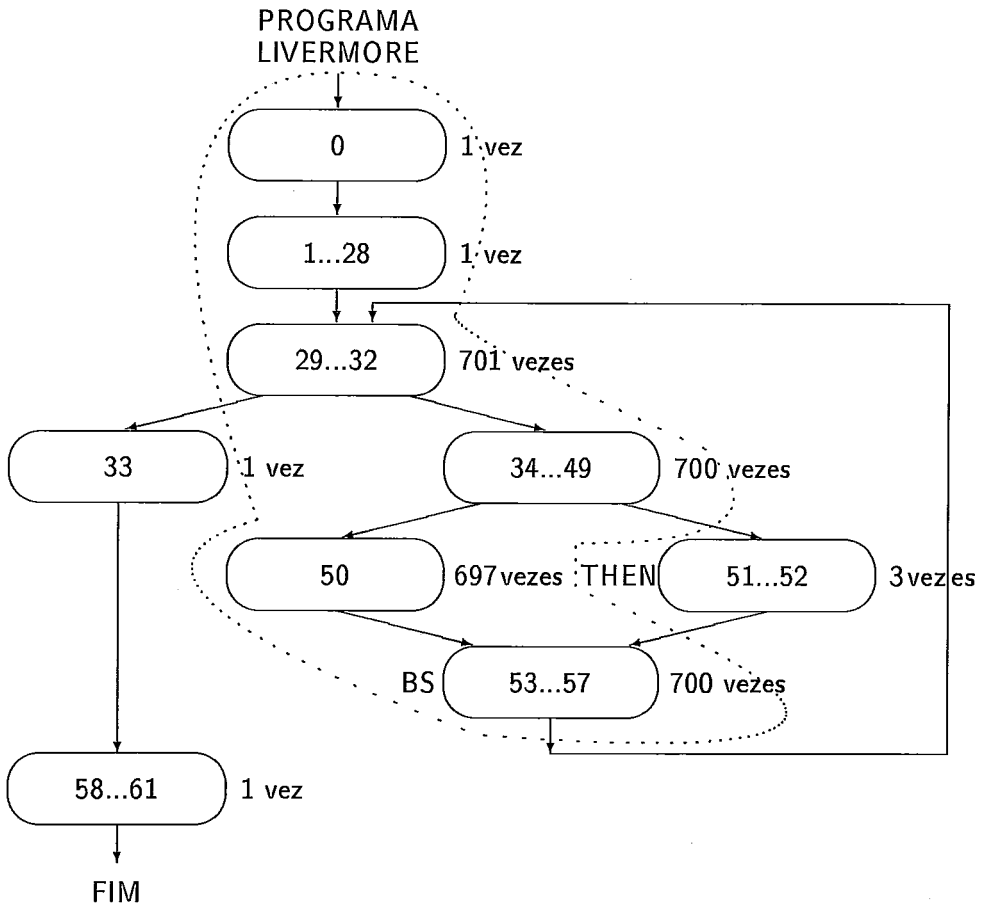


Figura 6.1: *Trace* do Programa *Livermore Loop 24*

Código para Execução na	Nº de Instruções		Nº de IMFs	
	TPE	TPE+	TPE	TPE+
Configuração 1	61	59	110	108
Configuração 2	61	59	74	64
Configuração 3	61	59	62	52

Tabela 6.1: Efeito do TPE e do TPE+ no Programa *Livermore Loop*

Na Tabela 6.1 vemos o número de instruções e o número de instruções longas (IMFs) de cada uma das formas executáveis obtidas para o programa *Livermore Loop*.

Na Tabela 6.1 podemos observar a redução no tamanho do programa exe-

cutável para cada uma das configurações propostas. Vemos que o programa *Livermore* o código obtido por meio TPE para qualquer das configurações do modelo CONDEX é formado por 61 instruções, e o obtido por meio do TPE+ por 59 instruções. O número de IMFs para as Configurações 1, 2 e 3 do programa executável resultante do TPE, é de 110, 74 e 62 respectivamente. Para o código oriundo do TPE+, esse número cai para 108 na Configuração 1, para 64 na Configuração 2, e para 52 na Configuração 3.

Código Executado na	Inst. Executadas		IMFs Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Configuração 1	17540	16143	29487	33664	1,167	1,021
Configuração 2	17540	16143	21063	19642	1,633	1,753
Configuração 3	17540	16143	21051	19630	1,634	1,752

Tabela 6.2: Efeito do TPE e do TPE+ na Execução do Programa *Livermore Loop*

O efeito do TPE e do TPE+ durante a interpretação do programa *Livermore Loop* número 24, é mostrado na Tabela 6.2. Na tabela observamos o número instruções e o número de IMFs executadas durante interpretação do programa de teste em cada uma das configurações do modelo CONDEX. Na terceira coluna da tabela temos a aceleração obtida na execução em relação a execução seqüencial¹.

Na Tabela 6.2 observamos que para o programa *Livermore Loop* submetido ao TPE, o número de instruções executadas nas Configurações 1, 2 e 3 é nos três casos 1740. Para o código executável obtido por meio do TPE+ esse número para as três configurações cai para 16143.

Examaminando ainda a Tabela 6.2 vemos que o número de ciclos de processador (i.e., IMFs) consumidos na interpretação do programa de teste, quando usamos escalonamento de instruções TPE é 29487, 21063 e 21051 respectivamente na Configuração 1, 2 e 3. Já quando o TPE+ é empregado esses valores são 33664, 19642 e 19630. Ou seja, o *speedup* é maior para o TPE na Configuração 1, e nas Configurações 2 e 3 é maior para o TPE+.

Observando o histograma contido na Figura 6.2, vemos uma outra forma de representar os resultados relativos ao tempos de execução obtidos. Quando comparamos o tempo gasto na execução seqüencial do programa *Livermore*, com o tempo

¹Os resultados da execução seqüencial são mostrados no Capítulo 5

gasto para a execução desse mesmo programa, compactado segundo as técnicas TPE e TPE+, temos que na Configuração 1 esse tempo cai para 85,70% e 97,84cai para 61,22% e 57,03%, e na Configuração 3 cai para 61,18% e 57,03%.

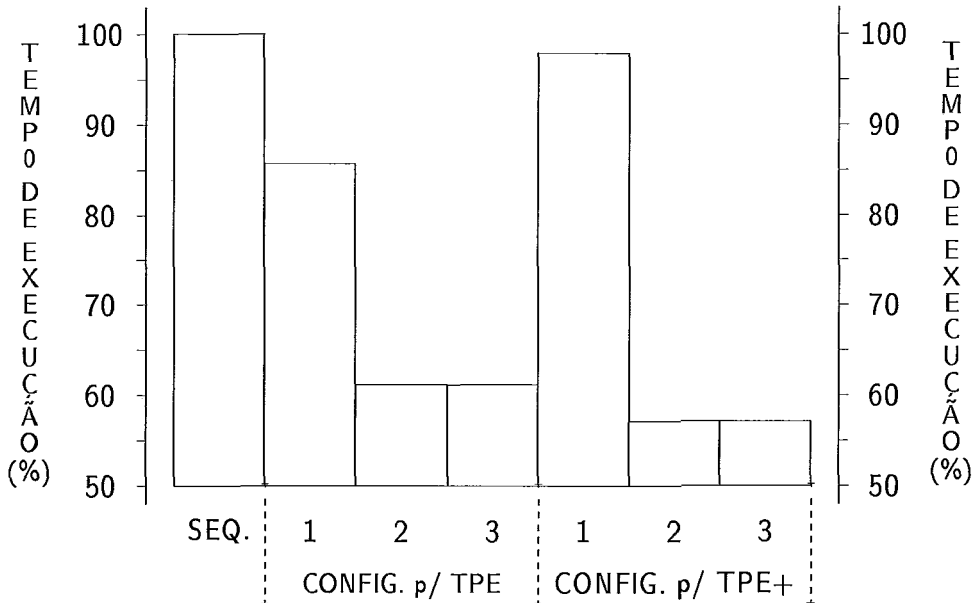


Figura 6.2: Execução Seqüencial × Execução TPE e TPE+ do Programa *Livermore Loop*

6.3.2 O Programa “Branch”

O *trace* escolhido no programa *Branch* (mostrado na Figura 6.3), para ser submetido ao TPE, possui seis blocos básicos. Um dos blocos básicos que pertencem ao *trace* selecionado é o bloco “be”, que inclui as instruções 29...33. Nesse caso o trecho poderia incluir o “be” (formado pelas instruções 34...37) já que a freqüência de execução desse bloco é a mesma que a do “bt.” Como podemos ver na figura ambos os blocos são executados 1280 vezes.

Observamos ainda na Figura 6.3, que da mesma forma que no programa *Livermore Loop*, o *trace* não inclui os dois blocos (“bt” e “be”) usados na compactação condicional descrita no Capítulo 5. Dessa forma, ao realizarmos o TPE+, os dois blocos, “bt” e “be,” já fundidos em um único bloco, passam a fazer parte do *trace* compactado.

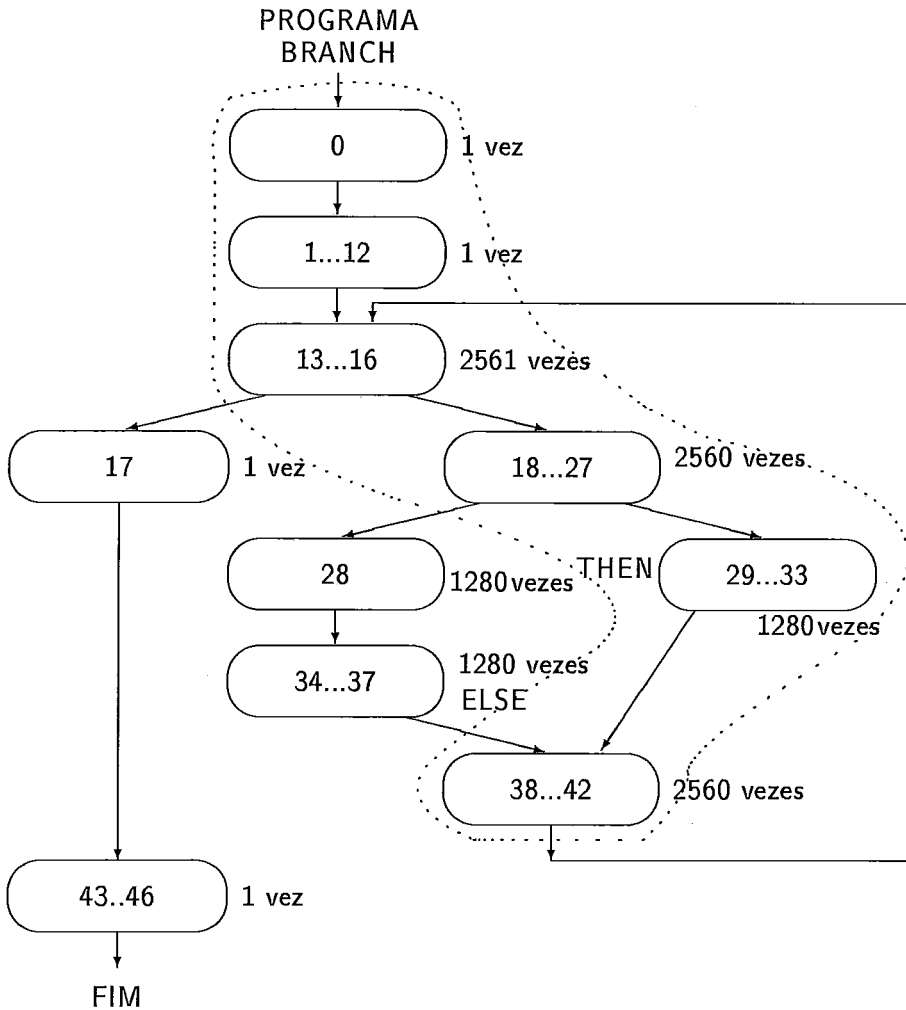


Figura 6.3: *Trace* do Programa *Branch*

Na Tabela 6.3 ilustramos o efeito sobre o tamanho do código do executável do programa de teste, quando as políticas TPE e do TPE+ são usadas para as três configurações do modelo CONDEX.

Observando a Tabela 6.3, vemos que as para o TPE e para o TPE+, na Configuração 1 temos 46 e 43 instruções no código executável. Para as Configurações 2 e 3, esse número é de 50 e 43 instruções respectivamente. O número de instruções aumenta nas Configurações 2 e 3, por causa da necessidade de duplicar o código do “bs” (instruções 38...42), para manter equivalência semântica do programa quando o “be” que não faz parte do *trace* for executado. Na Configuração 1, devido a restrição de recursos, durante a compactação do *trace* não foi possível movimentar as operações do “bs” para o “bt” não havendo portanto a necessidade da duplicação.

Depois de realizada a compactação segundo as políticas TPE e TPE+, o código executável em cada uma das configurações do simulador do CONDEX, apresenta para o TPE 83, 66, e 61 IMFs. Para o TPE+ o número de IMFs é de 71, 54 e 49 (Figura 6.3).

Código para Execução na	Nº de Instruções		Nº de IMFs	
	TPE	TPE+	TPE	TPE+
Configuração 1	46	43	83	71
Configuração 2	50	43	66	54
Configuração 3	50	43	61	49

Tabela 6.3: Efeito do TPE e do TPE+ no Programa *Branch*

Na Tabela 6.4 efeito das políticas TPE e TPE+ na interpretação do programa *Branch*. Vemos na tabela que o *speedup* atingido para as duas técnicas, na Configuração 1 é de 1,133 e 1,186. Quando dobramos o número de cada tipo das unidades funcionais do processador (Configuração 2), vemos que a aceleração atinge 1,645 e 1,500. Finalmente, quando mais uma vez dobramos o número de unidades funcionais da Configuração 2 e para obter a Configuração 3, a aceleração passa para 1,700 e 1,545.

Código Executado na	Inst. Executadas		IMFs Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Configuração 1	58901	53781	115234	110114	1,133	1,186
Configuração 2	57621	53781	79386	87066	1,645	1,500
Configuração 3	57621	53781	76822	84502	1,700	1,545

Tabela 6.4: Efeito do TPE e do TPE+ na Execução do Programa *Branch*

O histograma mostrado na Figura 6.4 ilustra o comportamento do tempo de execução do programa *Branch* em cada configuração do simulador CONDEX usada. Quando comparamos o tempo gasto na execução seqüencial do programa *Branch*, com o tempo gasto para a execução desse mesmo programa, compactado segundo as técnicas TPE e TPE+, temos que na Configuração 1 esse tempo cai para 88,23% e 84,31% para 60,78% e 66,66%, e na Configuração 3 cai para 58,82% e 64,70%.

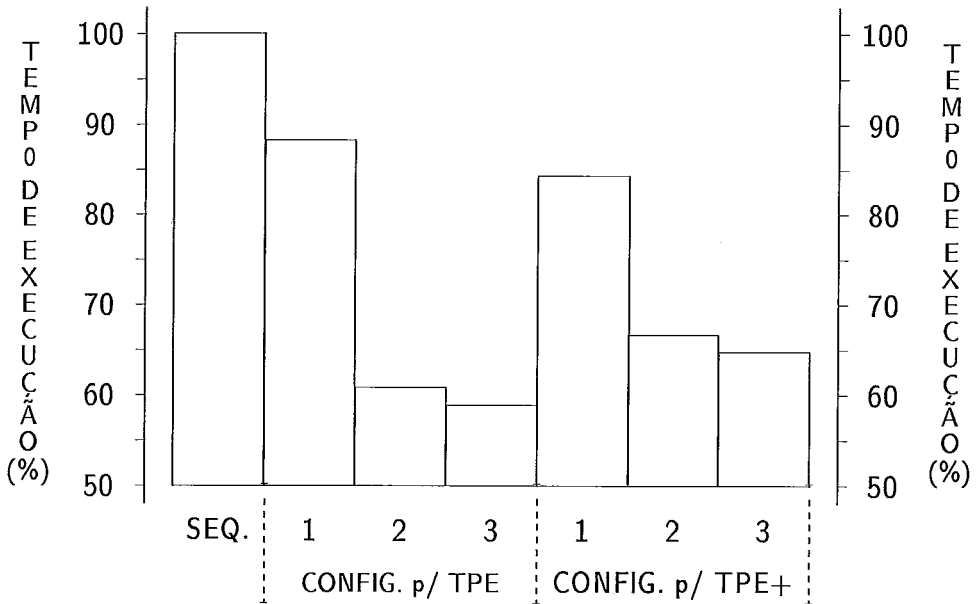


Figura 6.4: Execução Sequencial × Execução TPE e TPE+ do Programa *Branch*

Os resultados mostram que a política TPE se mostrou melhor que a TPE+ apenas na Configuração 1, para as duas outras configurações o uso TPE+ gerou código executável mais rápido que o TPE.

6.3.3 O Programa “BCD”

No caso da programa *BCD* mostrado na Figura 6.5, observamos que o trecho escolhido para ser submetido ao TPE, é formado por dez blocos básicos. Também nesse programa os dois blocos (“bt” e “bs”) envolvidos na compactação condicional, não fazem ambos, parte do trecho, mas somente o bloco “bs.”

O *trace* usado durante o processo de geração do código paralelo baseado no TPE+, inclui ambos os blocos “bt” e “bs,” empregados na compactação condicional. Vale lembrar que graças à compactação condicional, esses dois blocos agora compõem um único bloco.

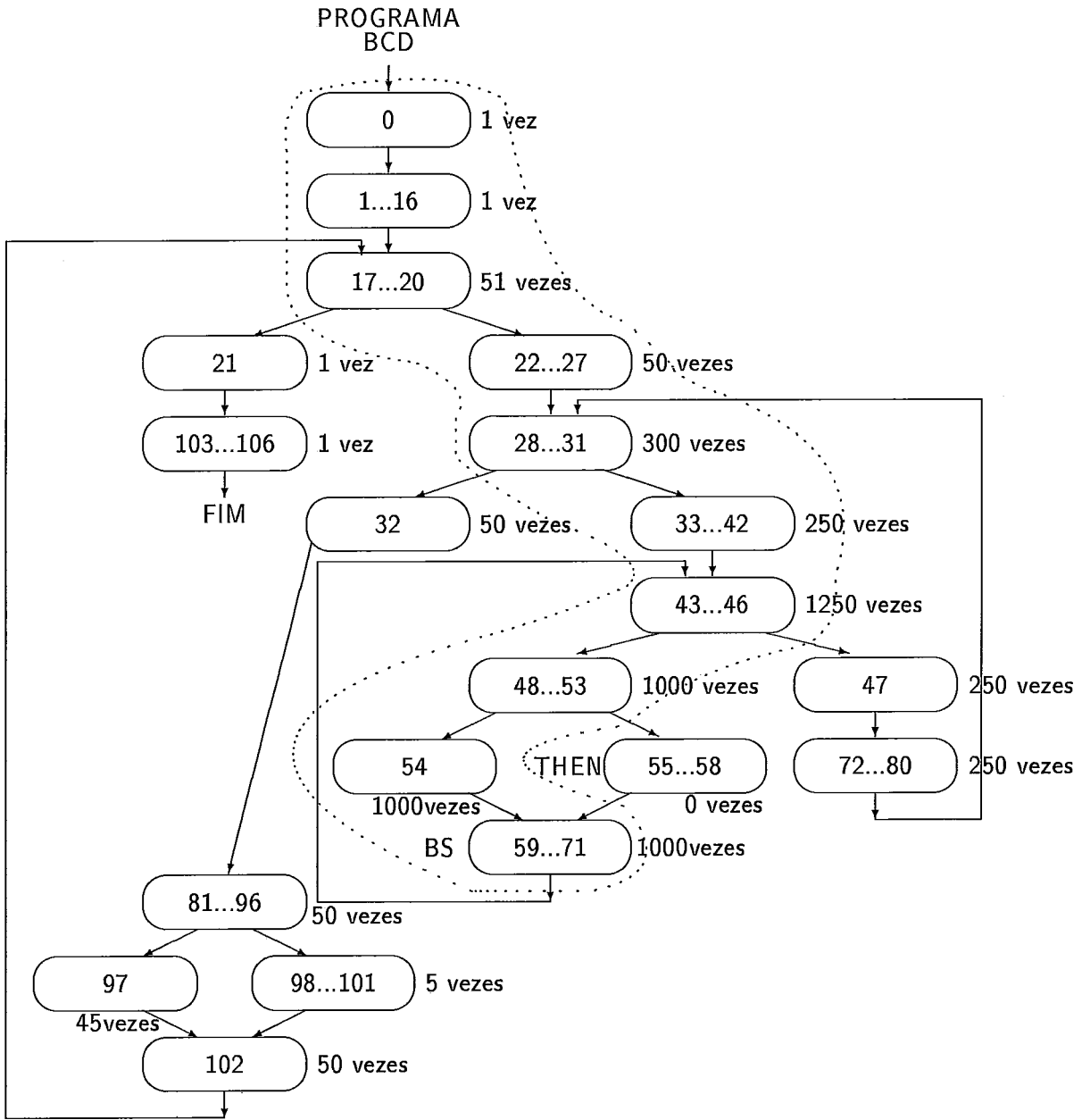


Figura 6.5: Trace do Programa *BCD*

O número de instruções e o número de IMFs de cada uma das formas executáveis obtidas por meio do TPE e do TPE+ para o programa *BCD*, são mostradas na Tabela 6.5. Para o código obtido por meio do TPE temos 104 instruções, e para o obtido por meio do TPE+ temos 102, para as três configurações da arquitetura. Os códigos executáveis gerados para as Configurações 1, 2 e 3, possuem respectivamente 195, 136, e 105 IMFs.

Código para Execução na	Nº de Instruções		Nº de IMFs	
	TPE	TPE+	TPE	TPE+
Configuração 1	104	102	197	195
Configuração 2	104	102	139	136
Configuração 3	104	102	112	105

Tabela 6.5: Efeito do TPE e do TPE+ no Programa *BCD*

Mostramos o efeito das técnicas compactação TPE e TPE+ durante a interpretação do programa *BCD*, na Tabela 6.6.

Código Executado na	Inst. Executadas		IMFs Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Configuração 1	31471	29598	61908	71382	1,175	1,019
Configuração 2	31471	29598	44336	49320	1,641	1,475
Configuração 3	31471	29598	34730	35714	2,095	2,037

Tabela 6.6: Efeito do TPE e do TPE+ na Execução do Programa *BCD*

Na Tabela 6.6 observamos que na Configuração 1 durante a execução do programa *BCD* quando a técnica de compactação TPE é usada, são necessários 61906 ciclos de processador. Para o TPE+ nessa configuração, o número de ciclos requerido é maior, isto é 71382 são necessários. Na Configuração 2 o tempo de execução do programa de teste para as duas técnicas é de 44336 ciclos para o TPE, e de 49320 ciclos para o TPE+. Finalmente na Configuração 3, são necessários 34730 e 35714 ciclos de processador, para que o programa *BCD* gerado respectivamente por meio do TPE e do TPE+, execute.

Uma outra forma de examinar os resultados da Tabela 6.6 é por meio da do histograma mostrado na Figura 6.6.

Observando o tempo gasto na execução do programa *BCD* obtido por meio do TPE, e o tempo gasto para a execução deste mesmo programa, obtido por meio do TPE+, vemos no histograma que na Configuração 1 esse tempo corresponde respectivamente 85,06% e 98,08% do tempo requerido para a execução seqüencial. Na Configuração 2 cai para 60,92% e 67,77%. Na Configuração 3 cai ainda mais para 47,72% e 49,07%. Nas três configurações o tempo de execução para o código

compactado por meio do TPE, é pior que o tempo de execução para o obtido por meio do TPE+.

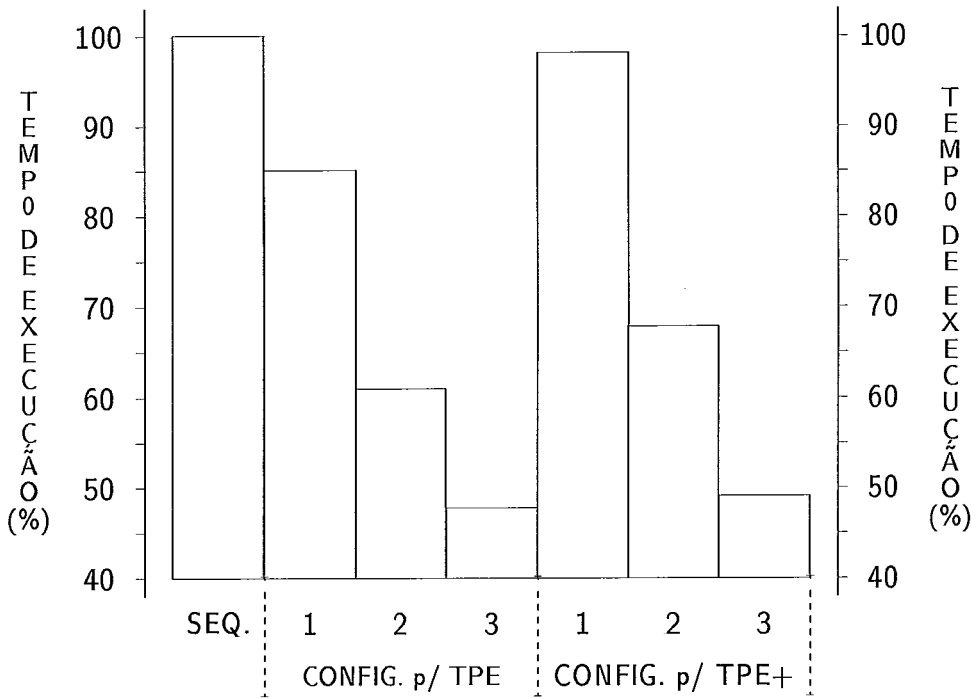


Figura 6.6: Execução Seqüencial × Execução TPE e TPE+ do Programa *BCD*

6.3.4 O Programa “Simpson”

Na Figura 6.7 mostramos o GFC do programa *Simpson* e na Figura 6.8 temos o GFC do procedimento *Simp* que o programa chama durante a sua execução. Não apresentamos nesse ponto, o procedimento *Fun* (que é chamado pelo procedimento *Inser* durante a interpretação do programa de teste), já que neste não existe nenhum *trace* para ser submetido ao TPE e ao TPE+. Uma visão completa do grafo do fluxo de controle do programa *Simpson*, faz parte do Capítulo 5.

No GFC da Figura 6.7 vemos o *trace* usado no TPE. Nesse caso o mesmo trecho é usado para o TPE+ já que nenhum bloco básico desse diagrama, foi submetido à compactação condicional. O trecho escolhido possui quatro blocos básicos.

No procedimento *Simp*, cujo GFC aparece na Figura 6.8, observamos que selecionamos dois *traces*. O primeiro deles é composto de seis blocos básicos, e o

segundo de três.

Para o TPE+ os dois trechos incluem os dois pares de blocos “bt” e “be” submetidos à compactação condicional. É conveniente lembrar que os dois pares de blocos ficam reduzidos a dois blocos devido por causa desse processo de compactação.

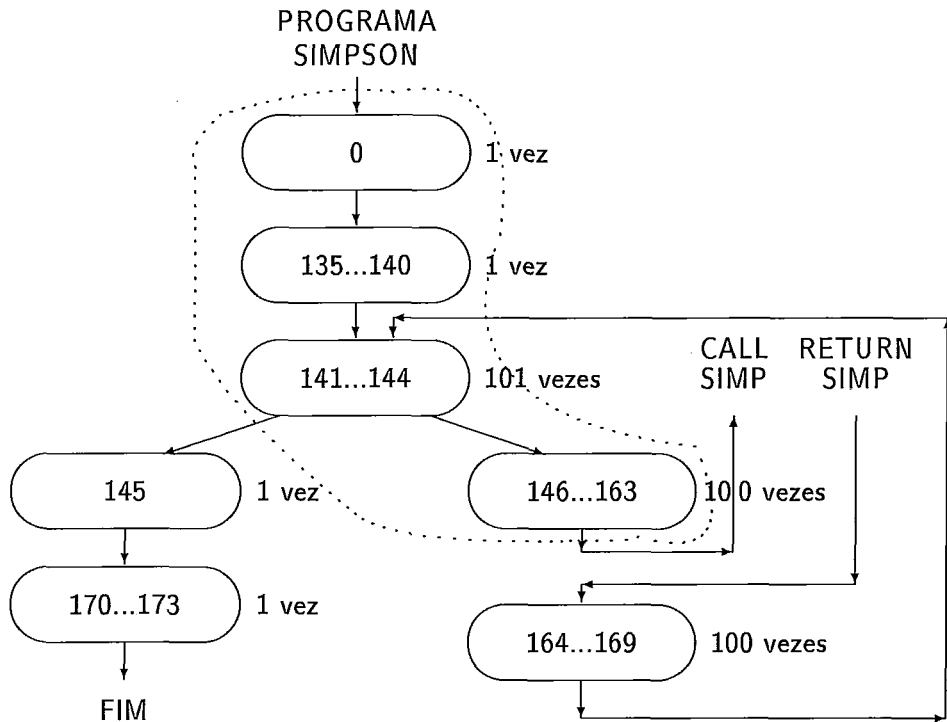


Figura 6.7: *Trace do Programa Simpson*

Observando a Tabela 6.7, vemos que depois de realizada as compactações TPE e TPE+, o código executável do programa *Simpson* apresenta respectivamente 170 e 168 instruções para Configuração 1. Já para a Configuração 2 do simulador CONDEX, o código é formado por 170 e 167 instruções, no caso do TPE e do TPE+. Na Configuração 3 o número de instruções dos códigos executáveis obtidos por meio das duas políticas de compactação usadas, é de 175 e 167 instruções respectivamente.

Os códigos executáveis obtidos para as três configurações, contêm 326, 225 e 206 IMFs no caso do TPE, e 304, 206 e 184 IMFS para o TPE+.

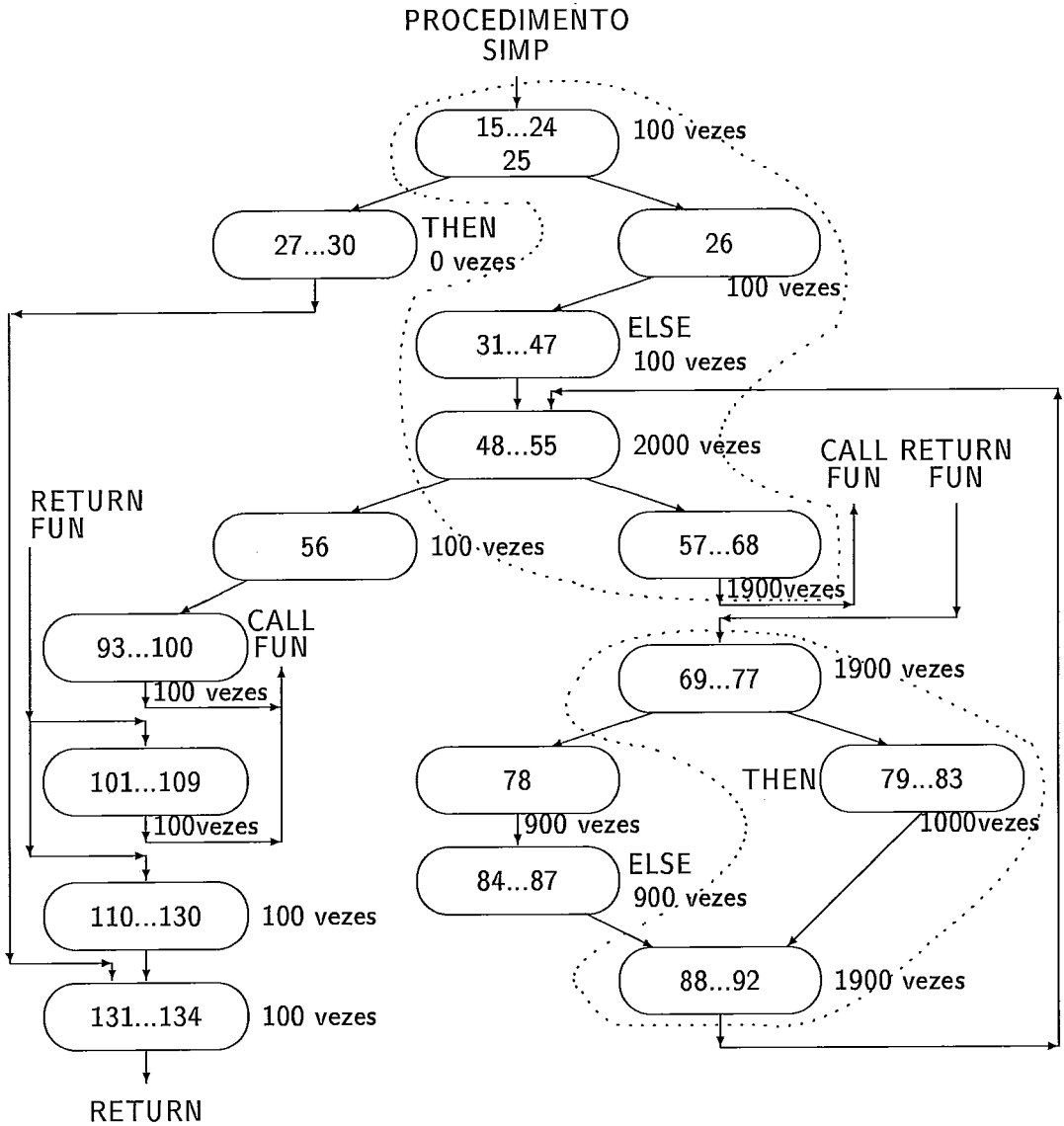


Figura 6.8: *Traces* do Procedimento *Simp* do Programa *Simpson*

Código para Execução na	Nº de Instruções		Nº de IMFs	
	TPE	TPE+	TPE	TPE+
Configuração 1	170	168	326	304
Configuração 2	170	167	225	206
Configuração 3	175	167	206	184

Tabela 6.7: Efeito do TPE e do TPE+ no Programa *Simpson*

Mostramos o efeito das técnicas de compactação TPE e TPE+, durante a

interpretação do programa *Simpson* na Tabela 6.8.

Código Executado na	Inst. Executadas		IMFs Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Configuração 1	110216	109215	223822	223622	1,205	1,206
Configuração 2	110212	108215	159617	159519	1,690	1,691
Configuração 3	111115	108215	141419	147519	1,908	1,829

Tabela 6.8: Efeito do TPE e do TPE+ na Execução do Programa *Simpson*

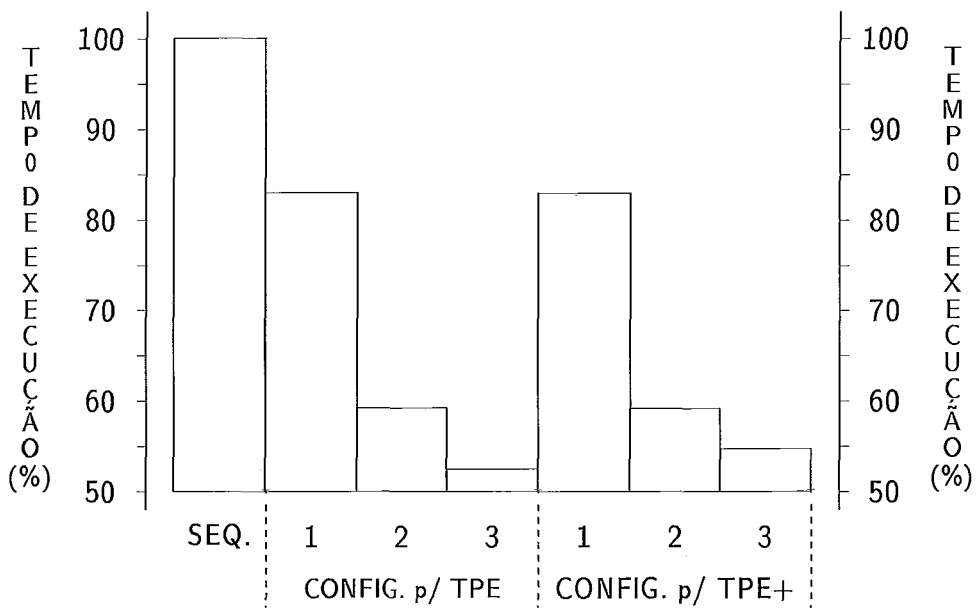


Figura 6.9: Execução Sequencial × Execução TPE e TPE+ do Programa *Simpson*

Examinando a Tabela 6.8, vemos que na Configuração 1 utilizada nos nossos experimentos, o número de ciclos de processador consumidos para a interpretação do programa de teste é praticamente igual para as duas técnicas de compactação, isto é 223822 e 223622. Para a Configuração 2 a situação se repete: 159617 IMFs foram executadas para o TPE e 159519 para o TPE+. Finalmente durante a execução dos códigos compactados com o TPE e o TPE+ para a Configuração 3, foram executadas 141419 e 147519 IMFs respectivamente.

No histograma da Figura 6.9 temos a representação gráfica dos valores da Tabela 6.8. A figura mostra que Configuração 1, para o TPE, o tempo de execução

corresponde a 82,94% do necessário para a execução seqüencial. Para o TPE+, nessa mesma configuração, corresponde a 82,87% desse tempo. Na Configuração 2, o número de ciclos consumidos na execução dos programas submetidos ao TPE e ao TPE+, cai respectivamente para 59,15% e 59,11% do tempo de execução seqüencial, e na Configuração 3 os dois tempos de execução avaliados, são 52,41% e 54,67% desse tempo.

6.3.5 O Programa “Árvore”

Finalmente na Figura 6.10 temos o GFC do procedimento *Inserer*, chamado pelo programa principal *Árvore* durante sua execução. Não repetimos aqui o GFC do programa principal apresentado no Capítulo 5, pois este não apresenta *traces* utilizados pelas técnicas TPE e TPE+.

Na Figura 6.10 observamos os dois trechos usados no TPE, delimitados por meio da curva tracejada. O primeiro trecho é formado por dez blocos básicos, e o segundo por oito. Obviamente nenhum dos dois *traces* incluem os dois blocos (“bt” e “be”) usados na compactação condicional descrita no Capítulo 5.

Para o TPE+ , os dois trechos incluem os quatro pares de blocos ‘bt’ e ‘be,’ submetidos à compactação condicional, isto é, cada par de blocos “bt be” já constitui um único bloco contendo instruções dos dois blocos originais.

O número de instruções e o número de IMFs de cada uma das formas executáveis obtidas para o programa *Árvore*, são mostradas na Tabela 6.9. Para o código obtido por meio do TPE para a Configuração 1 temos 186 instruções. Por outro lado, o código gerado pelo TPE+ nessa mesma configuração apresenta 180 instruções. Para as Configurações 2 e 3, esses números são 190 e 180 instruções para o TPE e o TPE+, respectivamente.

Na Tabela 6.9 temos que o número de IMFs do programa *Árvore* para o TPE e TPE+ é de 308 e 263 para o código executável na Configuração 1; de 234 e 189 para o executável na Configuração 2; e de 209 e 163 para o destinado à Configuração 3.

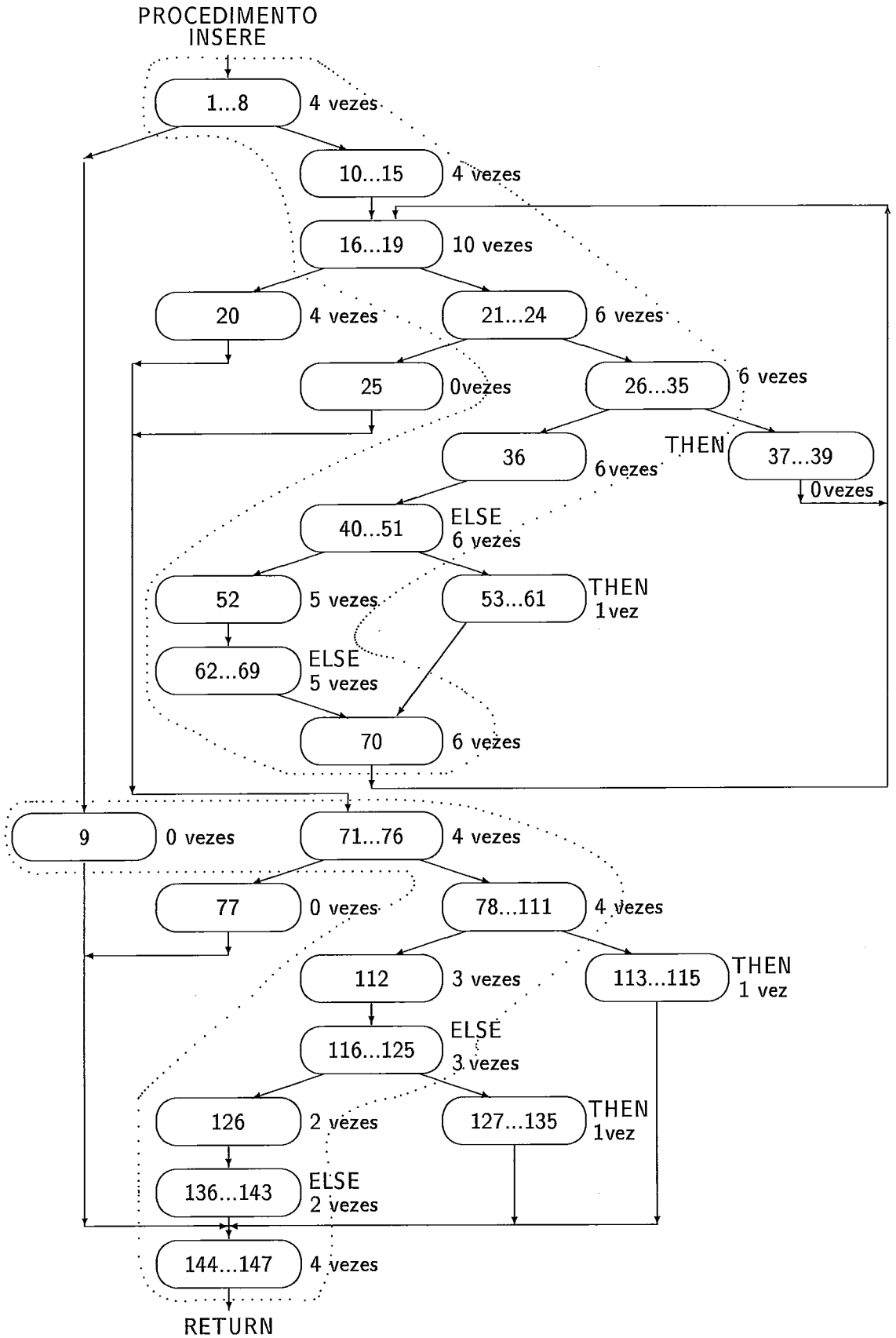


Figura 6.10: *Traces* do Procedimento *Inserere* do Programa *Árvore*

Código para Execução na	Nº de Instruções		Nº de IMFs	
	TPE	TPE+	TPE	TPE+
Configuração 1	186	180	308	263
Configuração 2	190	180	234	189
Configuração 3	190	180	209	163

Tabela 6.9: Efeito do TPE e do TPE+ no Programa *Árvore*

O efeito das políticas de compactação TPE e TPE+, durante a interpretação do programa *Árvore*, aparece nos resultados contidos na Tabela 6.10.

Código Executado na	Inst. Executadas		IMFs Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Configuração 1	565	547	1020	1001	1,216	1,239
Configuração 2	564	547	737	736	1,683	1,686
Configuração 3	564	547	654	645	1,897	1,924

Tabela 6.10: Efeito do TPE e do TPE+ na Execução do Programa *Árvore*

O número de instruções executadas para o programa de teste (para o TPE e para o TPE+), encontra-se na Tabela 6.10. Na Configuração 1, 565 e 547 instruções são executadas para os códigos provenientes do TPE e do TPE+. Nas Configurações 2 e 3, esse números são de 564 e 547 em ambos os casos.

Na Tabela 6.10 vemos que o número de IMFs executadas decresce a medida que são acrescentadas unidades funcionais na arquitetura. Durante a interpretação dos códigos produzidos por meio do TPE e do TPE+, são consumidos na Configuração 1, 1020 e 1001 ciclos; na Configuração 2, 737 e 736 ciclos; e na Configuração 3, 654 e 645 ciclos de Processador.

Para concluir, vemos no histograma da Figura 6.11 (para o programa *Árvore*), a comparação do tempo gasto na execução seqüencial com o tempo de necessário para a execução dos códigos resultantes do TPE e do TPE+ nas três configurações do modelo Condex usadas. Na Configuração 1 o tempo de execução para o TPE, cai para 82,19%, para o TPE+ cai para 80,66%. Já na Configuração 2 o tempo cai para 59,38% e 59,30% respectivamente para o TPE e para o TPE+. Finalmente na Configuração 3 cai para 52,69% e para 51,97%, como ilustra a figura.

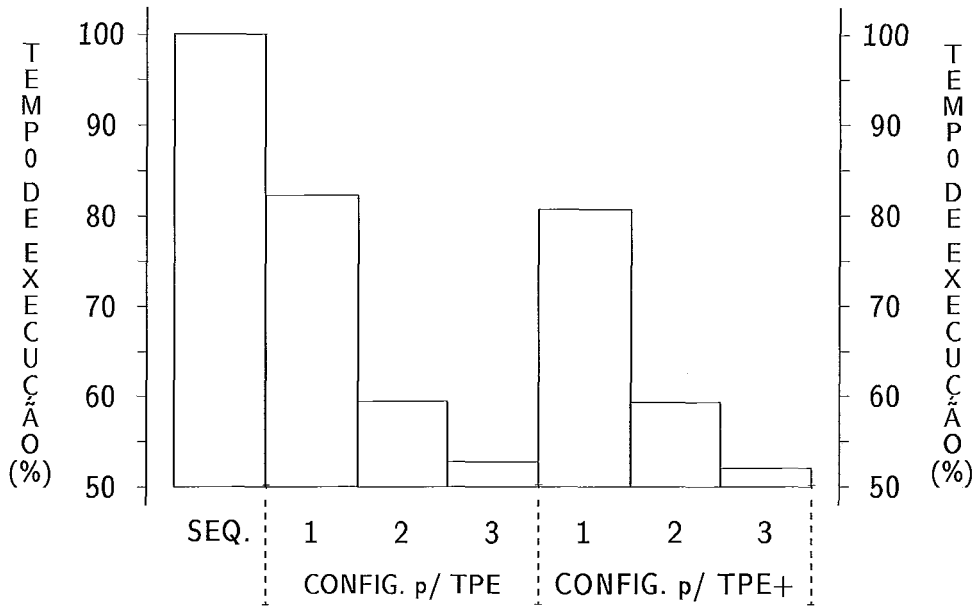


Figura 6.11: Execução Seqüencial × Execução TPE e TPE+ do Programa *Árvore*

6.4 Sumário dos Resultados Obtidos

Para possibilitar ao leitor uma visão global dos resultados obtidos, apresentamos nessa seção, seis tabelas que refletem o efeito da compactação condicional em cada programa de teste e a sua influência no tempo de execução desses.

Nas Tabelas 6.11, 6.12 e 6.13 apresentamos o número de IMFs e o número de instruções dos códigos executáveis de cada programa de teste submetido à compactação TPE e TPE+, para cada uma das configurações do modelo CONDEX.

Programa	Nº de IMFs		Nº de Instruções	
	TPE	TPE+	TPE	TPE+
Livermore	110	108	61	59
Branch	83	71	46	43
BCD	197	195	104	102
Simpson	326	304	170	168
Árvore	308	263	186	180

Tabela 6.11: N^{os} de IMFs e de Instruções Obtidos para a Configuração 1

Programa	Nº de IMFs		Nº de Instruções	
	TPE	TPE+	TPE	TPE+
Livermore	74	64	61	59
Branch	66	54	50	43
BCD	139	136	104	102
Simpson	225	206	170	167
Árvore	234	189	190	180

Tabela 6.12: N^{os} de IMFs e de Instruções Obtidos para a Configuração 2

Programa	Nº de IMFs		Nº de Instruções	
	TPE	TPE+	TPE	TPE+
Livermore	62	52	61	59
Branch	61	49	50	43
BCD	112	105	104	102
Simpson	206	184	175	167
Árvore	209	163	190	180

Tabela 6.13: N^{os} de IMFs e de Instruções Obtidos para a Configuração 3

Finalmente, os resultados dinâmicos da compactação, isto é, das técnicas TPE e TPE+, durante a interpretação dos programas teste, são apresentados nas Tabelas 6.14, 6.15 e 6.16. Essas tabelas além dos números de IMFs e instruções executadas, incluem o *speedups* obtidos para cada programa e o *speedup* médio observado em cada configuração.

Programa	Nº de IMFs Executadas		Nº de Instr. Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Livermore	29487	33664	17540	16140	1,167	1,021
Branch	115234	110114	58901	53781	1,133	1,186
BCD	61906	71382	31471	29598	1,175	1,019
Simpson	223822	223622	110216	109215	1,205	1,206
Árvore	1020	1001	565	547	1,216	1,239

Speedup médio: 1,179 1,134

Tabela 6.14: N^{os} de IMFs e Instruções Executadas na Configuração 1

Programa	Nº de IMFs Executadas		Nº de Instr. Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Livermore	21063	19642	17540	16143	1,633	1,753
Branch	79386	87066	57621	53781	1,645	1,500
BCD	44336	49320	31471	29598	1,641	1,475
Simpson	159617	159519	110212	108215	1,690	1,691
Árvore	737	736	564	547	1,683	1,686

Speedup médio: 1,658 1,621

Tabela 6.15: Nº^{os} de IMFs e Instruções Executadas na Configuração 2

Programa	Nº de IMFs Executadas		Nº de Instr. Executadas		Speedup	
	TPE	TPE+	TPE	TPE+	TPE	TPE+
Livermore	21051	19630	17540	16143	1,634	1,752
Branch	76822	84502	57621	53781	1,700	1,545
BCD	34730	35714	31471	29598	2,095	2,037
Simpson	141419	147519	111115	108215	1,908	1,829
Árvore	654	645	564	547	1,897	1,924

Speedup médio: 1,846 1,817

Tabela 6.16: Nº^{os} de IMFs e Instruções Executadas na Configuração 3

No Capítulo 8, apresentamos o resumo do trabalho de desenvolvido e as sugestões para pesquisas futuras.

Capítulo 7

Conclusões

Nesse trabalho apresentamos o estudo sobre o efeito da execução condicional, em um modelo de arquitetura Super Escalar do tipo *Very Large Instruction Word* que chamamos CONDEX. O modelo proposto permite a execução condicional de instruções contidas em uma mesma instrução multifuncional. Essa capacidade permite que uma dada instrução longa contenha instruções oriundas de blocos básicos distintos.

7.1 Resumo do Trabalho

No Capítulo 2 fizemos uma apresentação das características de arquiteturas Super Escalares e mostramos que o nível de implementação (*hardware* ou *software*), do algoritmo responsável pela detecção do paralelismo de programas, é usado para distinguir a classe que o processador pertence. Mostramos as técnicas de detecção de paralelismo e como as restrições impostas pelas dependências de dados presentes no código, e pela limitação de recursos disponíveis nos processadores, interferem no processo da extração da concorrência presente entre as instruções do código seqüencial. Finalmente descrevemos de que forma a latência de instruções restringem o desempenho dessas arquiteturas.

No Capítulo 3 examinamos aspectos relativos ao modelo de processador Super Escalar Condex, que permite execução condicional de operações empacotadas em instruções do programa executável. Inicialmente examinamos o código intermediário seqüencial que deve ser paralelizado durante o processo de compactação e mostramos a ação de cada operação que pode ser gerada durante o processo de compilação. Descrevemos o modelo da VLIW proposto, isto é, seus componentes, a interligação

existente entre os mesmos e a latência de cada uma das suas unidades funcionais. Ainda nessa seção, mostramos como simulamos o modelo de arquitetura proposto.

No Capítulo 4 mostramos os passos necessários para a geração do código paralelo que é interpretado no simulador do modelo CONDEX. Examinamos inicialmente a fase em que o código intermediário seqüencial é submetido á compactação local. Por fim descrevemos como é realizada a compactação condicional.

No Capítulo 5 descrevemos a metodologia empregada nos nossos experimentos, mostramos o *benchmark* usado na avaliação de desempenho, e as diferentes configurações do simulador do CONDEX testadas. Contabilizamos, para cada uma das configurações do processador utilizadas, o efeito estático resultante do emprego da compactação condicional, isto é, verificamos como a compactação condicional reduziu o número de instruções e instruções longas de cada programa teste. O efeito dinâmico da compactação condicional foi medido posteriormente durante interpretação dos programas teste, em cada uma das configurações e obtivemos um *speedup* no tempo de execução dos programas teste de até 1,95.

Finalmente no Capítulo 6, apresentamos duas técnicas de compactação baseadas no perfil de execução dos programas, que fazem uso da capacidade de execução condicional do nosso modelo de arquitetura. Nesse capítulo, descrevemos as características dessas duas modalidades escalonamento de instruções e apresentamos resultados obtidos pela compactação do código seqüencial e durante interpretação dos programas de testes. A utilização das duas políticas de escalonamento de instruções nos programas de teste, resultou em *speedup* de até 2,09.

Apresentamos no Apêndice A o código fonte dos nossos programas de teste e no Apêndice B, mostramos que o potencial de processamento oferecido pela inclusão de múltiplas unidades funcionais na arquitetura nem sempre é acompanhado pelo aumento proporcional no desempenho. Na prática a ativação concorrente dos recursos *hardware* presentes no processador nem sempre é possível por causa da ausência do paralelismo de baixo nível do programa testado. Obtivemos então o perfil da ocupação das diversas unidades funcionais presentes nas três configurações avaliadas do modelo proposto. Esses dados são úteis para o dimensionamento de um modelo de processador mais eficiente em termos da relação “custo × desempenho.”

7.2 Sugestões para Pesquisas Futuras

Sem sombra de dúvida a automatização da segunda etapa do processo de compactação, atualmente realizada manualmente, é de grande importância na continuidade das pesquisas sobre a execução condicional. A automatização irá possibilitar a geração de código executável mais rapidamente, permitindo levar a cabo um número maior de experimentos, isto é, será possível interpretar um número maior de programas teste em um conjunto mais amplo de configurações do CONDEX.

Esperamos com esses novos experimentos atingir resultados de desempenho ainda mais significativos, além da obtenção de mais dados para estabelecer o critério para a compactação das operações de blocos do tipo Bloco Sucessor em instruções que contém operações de blocos THEN, de modo a garantir a eliminação da anomalia constatada durante a nossa pesquisa e relatada no Capítulo 5.

Posteriormente sugerimos a expansão do processo de compactação condicional, de modo que instruções oriundas de mais de dois blocos básicos sejam compactadas em IMF's comuns, isto é, sugerimos escalonar em instruções multifuncionais, instruções cuja execução seja dependente do conteúdo de diferentes conjuntos de indicadores de condição. O modelo de arquitetura CONDEX que está equipado de diversos desses conjuntos de indicadores de condição possibilita essa alternativa.

Paralelamente com o aprimoramento do processo de compactação condicional, é importante continuar os experimentos com as técnicas baseadas no perfil de execução dos programas (TPE e TPE+), que num primeiro momento produziram *speedups* médios mais elevados que os obtidos por meio da compactação condicional.

Uma vez superadas essas etapas, pode ser útil o emprego de um compilador comercial para produção do código seqüencial a ser compactado, e concomitantemente nova simulação do modelo CONDEX adaptada ao novo conjunto ampliado de instruções.

Para finalizar, gostaríamos de observar que o desenvolvimento de compiladores que geram código seqüencial, mas que levem em conta uma possível extração de concorrência do código produzido, serão de grande importância para o aumento do desempenho das arquiteturas Super Escalares de ambas as classes (i.e., instruções escalonadas estática e dinamicamente). Um compilador criado sob essa ótica, poderia por exemplo manter os valores de variáveis em registradores, provocando uma

redução no número de acessos a memória, deixando para armazenar o conteúdo de qualquer registrador no momento que esse precisasse ser reutilizado.

Bibliografia

- [ACOS86] R. D. Acosta, J. Kjelstrup, and H. C. Torng, “An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors,” *IEEE Transactions on Computers*, Vol. 35, No. 9, September 1986, pp. 815–828.
- [AHOU86] A. V. Aho, R. Sethi and J. D. Ullman, “Compilers Principles, Techniques, and Tools,” Addison-Wesley Publishing Company, 1986.
- [ANAN90] K. Anantha and F. Long, “Code Compaction for Parallel Architectures,” *Software–Practice and Experience*, Vol. 20, No. 6, June 1990, pp. 537–534.
- [ANDE67] D. W. Anderson, F. J. Sparatio, and R. M. Tomasulo, “The IBM System/360 Model 91: Machine Philosophy and Instruction Handling,” *IBM Journal*, No. 11, 1967, pp. 2–7.
- [AMOR88] C. L. de Amorim, V. C. Barbosa e E. S. T. Fernandes, “Uma Introdução à Computação Paralela e Distribuída,” VI Escola de Computação, 1988.
- [AUHT85] Augustus K. Uht, “Hardware Extraction of Low Level Cocurrency from Sequential Instruction Streams,” Ph.D. Thesis, Carnegie-Mellon University, December 1985.
- [BAKO90] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye, “The IBM RISC System/6000 Processor: Hardware Overview,” *IBM Journal of Research and Development*, Vol. 34, No. 1, January 1990, pp. 12–22.
- [BARB93] Fernando M. B. Barbosa, “Efeito do Escalonamento Dinâmico no Desempenho de Processadores Super Escalares,” Tese de Mestrado, COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, 1993.

- [BORN91] Claudson F. Bornstein, Claudia Marcia D. Pereira e Edil S. T. Fernandes, "Experimentos com a Extração de Paralelismo," Relatório Técnico ES-243/91, Programa de Sistemas e Computação, COPPE/UFRJ, Março de 1991, pp. 1-16.
- [BUTL91] M. Butler, T. Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism is Greater than Two," Proceedings of the 18th Annual International Symposium on Computer Architecture, ACM and IEEE Computer Society, Toronto, Canada, May 1991, pp. 276-286.
- [CHAN91] Pohua P. Chang, William Y. Chen, Scott A. Mahlke, and Wen-mei W. Hwu, "Comparing Static and Dynamic Code Scheduling for Multiple Instruction Issue Processors," Proceedings of the 24th Annual International Symposium on Microarchitecture, MICRO-24, November 1991, pp. 25-33.
- [CLYM94] J. Clyman, "PowerPC: Your Next CPU?," PC Magazine, February 1994, pp. 181-197.
- [COLW87] R. P. Colwell, et al., "A VLIW Architecture for a Trace Scheduling Compiler," ASPLOS-II, ACM, 1987.
- [COLW88] R. P. Colwell, R. P. Nix, J. J. O' Donnell, D. B. Papworth and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," IEEE Transactions on Computers, Vol. 37, No. 8, August 1988, pp. 967-979.
- [CONT65] S. D. Conte, "Elementary Numerical Analysis," McGraw-Hill Book Co., New York, NY, USA, 1965.
- [DAVI90] J. W. Davidson and D. B. Whalley, "Reducing the Cost of Branches by Using Registers," The 17th Annual International Symposium on Computer Architecture, Washington, 1990, pp. 182-191.
- [DEC92] Digital Equipment Corporation, "Alpha Architecture Handbook - Preliminary Edition," DEC, 1992.
- [DIEF92] K. Diefendorff and M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," IEEE Micro, April 1992, pp. 40-63.

- [DITZ87] David R. Ditzel, e Hubert R. Mclellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proceedings of the 14th Annual International Symposium on Computer Architecture, 1987, pp. 2–9.
- [EBCI88] K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential–Natured Software," Proceedings IFIP, 1988.
- [ELLI86] John R. Ellis, "Bulldog: A Compiler for VLIW Architectures," ACM Doctoral Dissertation Award 1985, The MIT Press, USA, 1986.
- [FERN92] Edil S. T. Fernandes e Anna Dolejsi Santos, "Arquiteturas Super Escalares: Detecção e Exploração do Paralelismo de Baixo Nível," VIII Escola de Computação, 1992.
- [FERN92a] Edil S. T. Fernandes and Fernando Mauro B. Barbosa, "Effects of Building Blocks on the Performance of Super–Scalar Architectures," Proceedings of the 19th Annual International Symposium on Computer Architecture, IEEE Computer Society and ACM, May 1992, 10 pages.
- [FERN92b] Edil S. T. Fernandes, Claudson F. Bornstein, and Claudia M. D. Pereira, "Parallel Code Generation for Super–Scalar Architectures," Microprocessing and Microprogramming, The Euromicro Journal, North Holland, Vol.34, No. 1–5, February 1992, pp. 223–226.
- [FISC88] C. N. Fischer and R. J. LeBlanc Jr., "Crafting a Compiler," The Benjamin/Cummings Publishing Company, Inc., 1988.
- [FISH79] J. A. Fisher, "The Optimization of Horizontal Microcode within and beyond Basic Blocks: An Application of Processor Scheduling with Resources," U. S. Department of Energy Report COO–3077–161, Courant Mathematics and Laboratory, New York University, October 1979.
- [FISH81] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981, pp. 478–490.
- [FISH83] Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," Proceedings of the 10th Annual International Symposium on Computer Architecture, IEEE Computer Society and ACM, June 1983, pp. 140–150.

- [FISH84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, Published as Sigplan Notices, Vol. 19, No. 6, June 1984, pp. 37-47.
- [FISH84a] Joseph A. Fisher, "VLIW Machine: A Multiprocessor for Compiling Scientific Code," Computer, July 1984, pp. 45-53.
- [FLYN66] M. J. Flynn, "Very High-Speed Computers," Proceedings of the IEEE 54, December 1966, pp. 1901-1909.
- [GRAY94] S. Gray and R. Adams, "Using Conditional Execution to Exploit Instruction Level Concurrency," Technical Report no. 181, School of Information Sciences, Division of Computer Science, University of Hertfordshire, March 1994.
- [GROS82] T. R. Gross and J. L. Hennessy, "Optimizing Branch Delays," Proceedings of the 15th Annual International Workshop on Microprogramming and Microarchitecture, Palo Alto, CA, USA, 1982, pp. 114-120.
- [GROH90] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," IBM Journal of Research and Development, Vol. 34, No. 1, January 1990, pp. 37-58.
- [HENN81] J. Hennessy et al., "MIPS: A VLSI Processor Architecture," Proc. CMU Conf. VLSI Systems and Computations, Computer Science Press, Rockville, MD October 1981, pp. 337-346.
- [HENN83] J. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983, pp. 422-448.
- [HEST90] P. D. Hester, "RISC System/6000 Hardware Background and Philosophies," IBM RISC System/6000 Technology, IBM Corp., SA 23-619, January 1990, pp. 2-7.
- [HINT89] G. Hinton, "80960 - Next Generation," Proceedings of the 34th COMPCON, IEEE, San Francisco, CA, USA, March 1989, pp. 13-17.

- [HORD82] R. M. Hord, "ILLIAC IV: The First Supercomputer," Computer Science Press, Rockville, Md, USA, 1982.
- [HSIN93] Hsing Tse Hao, "Efeito da Predição de Desvios e da Interrupção Precisa no Desempenho de Processadores Super Escalares," Tese de Mestrado, COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, 1993.
- [HSU86] Peter Y. T. Hsu, "Highly Concurrent Scalar Processing," Ph.D. Thesis, University of Illinois at Urbana-Champaign, January 1986.
- [HSU86a] Peter Y. T. Hsu and Edward S. Davidson, "Highly Concurrent Scalar Processing," Proceedings of the 13th Annual International Symposium on Computer Architecture, IEEE Computer Society and ACM, June 1986, pp. 386–395.
- [INTE89] Intel, "Intel 80960CA User's Manual," Intel, 1989.
- [INTE89a] Intel, "i860 64-bit Microprocessor Programmer's Reference Manual," Intel, 1989.
- [JLEE84] J. K. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, Vol. 17, No. 1, January 1984, pp. 6–22.
- [JOUP89] N. Jouppi and D. Wall, "Available Instruction Level Parallelism for Super-Scalar and Super-Pipelined Machines," Proceedings of the Third ASPLOS (April 1989), in ACM SIGPLAN Notices, Vol. 14, May 1989.
- [KANE92] G. Kane and J. Heinrich, "MIPS RISC Architecture," Prentice-Hall, 1992.
- [KELL75] R. M. Keller, "Look-Ahead Processors," Computing Surveys, Vol. 7, No. 4, December 1975, pp. 177–195.
- [KRIC91] R. F. Krick and A. Dollas, "The Evolution of Instruction Sequencing," IEEE Computer, April 1991, pp. 5–15.
- [KOHN89] Les Kohn and Neal Margulis, "Introducing the Intel i860 64-Bit Microprocessor," IEEE Micro, Vol. 9, No. 4, August 1989, pp. 15–30.

- [LAM92] M. Lam and R. Wilson, "Limits of Control Flow on Parallelism," Proceedings of the 19th International Symposium on Computer Architecture, ACM and IEEE Computer Society, May 1992, pp. 46–57.
- [LAND80] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallet, "Local Microcode Compaction Techniques," Computing Surveys, Vol. 12, No. 3, September 1980, pp. 261–294.
- [LILJ88] David J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," IEEE Computer, Vol. 21, No. 7, July 1988, pp. 47–55.
- [MCFA86] Scott McFarling and John Hennessy, "Reducing the Cost of Branches," Proceedings of the 13th International Symposium on Computer Architecture, ACM and IEEE Computer Society, Tokyo, Japan, June 1986, pp. 396–403.
- [MCGE90] Steve Mcgeady, "Inside Intel's i960CA Superscalar Processor," Microprocessors and Microsystems, Vol. 14, No. 6, July/August 1990, pp. 385–396.
- [MCMA83] F. H. McMahon, "Fortran Kernels: MFLOPS," Lawrence Livermore National Laboratory, 1983.
- [MONT93] Leonardo Cardoso Monteiro, "Geração de Código Paralelo para Máquinas Super Escalares," Projeto de Fim de Curso, IM, UFRJ, 1993.
- [MOON92] S. Moon and K. Ebcioglu, "An Efficient Recourse-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," Proceedings of the 25th Annual International Workshop on Microprogramming and Microarchitecture MICRO-25, ACM and IEEE Computer Society, December 1992, pp. 55–71.
- [NICO85] A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," Technical Report TR-85-678, Department of Computer Science, Cornell University, May 1985.
- [NICO90] A. Nicolau and R. Potasman, "Realistic Scheduling: Compaction for Pipelined Architectures," Proceedings of the 23rd Annual International Workshop on Microprogramming and Microarchitecture MICRO-23, ACM and IEEE Computer Society, November 1990, pp. 69–79.

- [PATT85] Yale N. Patt, Wen-mei Hwu, and Michael C. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," Proceedings of the 18th International Microprogramming Workshop, December 1985, pp. 103–108.
- [PLES88] A. R. Pleszkun and G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors," Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp. 37–44.
- [PNEV94] D. N. Pnevmatikatos and G. S. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processors," Proceedings of the 21st Annual International Symposium on Computer Architecture, 1994, pp. 120–129.
- [RAU89] B. R. Rau, et al., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs," Computer, Vol. 22, No. 1, January 1989, pp. 12–35.
- [RAU93] B. R. Rau, "Dynamically Scheduled VLIW Processors," Proceedings of the 26th Annual International Workshop on Microprogramming and Microarchitecture MICRO-26, ACM and IEEE Computer Society, December 1993, pp. 80–92.
- [RYAN93] B. Ryan, "RISC Drives PowerPC," Byte, August 1993, pp. 79–90.
- [SALI76] Allan B. Salisbury, "Microprogrammable Computer Architectures," American Elsevier Publishing Co., Inc., USA, 1976.
- [SHUS77] L. J. Shustek, "Analysis and Performance of Computer Instruction Sets," Ph.D. Thesis, Stanford University, 1977.
- [SITE93] R. L. Sites, "Alpha AXP Architecture," Communications of the ACM, February 1993, pp. 33–44.
- [SMIT81] J. E. Smith, "A Study of Branch Prediction Strategies," The 8th Annual International Symposium on Computer Architecture, 1981, pp. 135–148.
- [SMOT93] M. Smotherman, S. Chawla, S. Cox and B. Malloy, "Instruction Scheduling for the Motorola 88110," Proceedings of the 26th Annual International Workshop on Microprogramming and Microarchitecture MICRO-26, ACM and IEEE Computer Society, December 1993, pp. 257–262.

- [SOUZ93] Alberto F. de Souza, "Avaliando os Parâmetros de uma Arquitetura VLIW," Tese de Mestrado, COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, 1993.
- [STAM94] N. Stam, "Looking Inside The PowerPC 601," PC Magazine, February 1994, pp. 199–207.
- [STEV94] F. L. Steven, G. B. Steven and L. Wang, "An Evaluation of the iHARP Multiple Instruction Issue Processor," Euromicro 94, September, 1994.
- [STON87] H. S. Stone, "High-Performance Computer Architecture," Addison-Wesley, 1987.
- [TABA91] Daniel Tabak, "Advanced Microprocessors," Mc Graw-Hill, Inc., USA, 1991.
- [THOM93] T. Thompson, "Power," Byte, August 1993, pp. 56–74.
- [THOR64] James E. Thornton, "Parallel Operation in the Control Data 6600," Proceedings of the Fall Joint Computer Conference, AFIPS, pt. 2, Vol. 26, 1964, pp. 33–40.
- [TJAD70] G.S. Tjaden and M. Flynn, "Detection and Parallel Execution of Independent Instructions," IEEE Transactions on Computers, Vol. C-19, No. 10, October 1970, pp. 889–895.
- [TJAD73] G.S. Tjaden and M. Flynn, "Representation of Concurrency with Ordering Matrices," IEEE Transactions on Computers, Vol. C-22, No. 8, August 1973, pp. 752–761.
- [TOKO78] M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "A Technique of Global Optimization of Microprograms," Proceedings of the 11th Annual Microprogramming Workshop, 1978, pp. 41–50.
- [TOKO81] M. Tokoro, E. Tamura and T. Takizuka, "Optimization of Microprograms," IEEE Transactions on Computer, Vol. C.30, No. 7, July 1981, pp. 491–504.
- [TOMA67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal, No. 11, 1967, pp. 25–33.

- [WALL91] D. Wall, "Limits of Instruction-Level Parallelism," Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 176–186.
- [WEIS84] Shlomo Weiss and James E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," Proceeding of the 11th Annual International Symposium on Computer Architecture, June 1984, pp. 110–118.
- [WIRT76] N. Eirth, "Algorithms + Data Structures = Programs," Prentice-Hall, Inc., 1976.
- [WOOD78] G. Wood, "On the Packing of Micro-Operations into Micro-Instruction Words," Proceedings of the 11th Annual Workshop on Micro-programming, SIGMICRO Newsletter, Vol. 9, No. 4, December 1978, pp. 51–55.

Apêndice A

Programas de Teste

Apresentamos nas Figuras A.1, A.2, A.3 A.4 e A.5 a seguir, o código Pascal dos programas utilizados nos nossos experimentos.

```
program Branch;
var
  i, odd, even, sum :integer;
begin
  sum := 0;
  odd := 0;
  even := 0;
  i := 0;
  while i < 2560 do
    begin
      sum := sum + i;
      if (i and 1) = 1 then odd := odd + 1
        else even := even + 1;
      i := i + 1;
    end;
end.
```

Figura A.1: O Programa *Branch*

```
program Livermor;
const
  N = 700;
VAR
  m,k : INTEGER;
  x : ARRAY[1..N] OF INTEGER;
begin
  x[1] := 680;
  x[2] := 691;
  x[3] := 692;
  x[4] := 683;
  x[5] := 674;
  x[6] := 665;
  x[7] := 676;
  x[8] := 677;
  x[9] := 668;
  x[10] := 699;
  m := 1;
  k := 1;
  while k <= N do
    begin
      if x[k] < x[m] then m := k;
      k := k + 1;
    end;
  end.
end.
```

Figura A.2: O Programa *Livemore*

```

program bcd;
var
  w : array[1..5] of integer;
var
  mask,sum,i,j,wgh,entr,res : integer;
begin
  entr := 1;
  w[1] := 1;
  w[2] := 10;
  w[3] := 100;
  w[4] := 1000;
  w[5] := 10000;
  while entr <= 80 do begin
    mask := 1;
    sum := 0 ;
    i := 1;
    while i <= 5 do
      begin
        wgh := w[i];
        j := 0 ;
        while j <= 3 do
          begin
            if ( (entr and mask ) <> 0 ) Then sum := sum + wgh;
            wgh := wgh * 2;
            mask := mask * 2;
            j := j + 1;
          end;
          j := j - 1;
          i := i + 1;
        end;
      res := sum;
      entr := entr + 1;
      if (sum - 10*(sum div 10)) = 9 then entr := entr + 6;
    end;
  end.

```

Figura A.3: O Programa *BCD*

```

program Simpson;
procedure Func(X : real; var Y : real);
{ -Funcao de integracao. }
begin
Y := X * X * X;
end;
procedure Simpson(LowerLimit : real; UpperLimit : real;
                  NumIntervals : integer; var Integral : real;
                  var Error : boolean);
{ -Calcula integral pelo metodo de Simpson. }
var
Spacing          : real;
Point            : real;
OddSum           : real;
EvenSum          : real;
LimitsValue      : real;
Interval         : integer;
Y, YUpper, YLower : real;
begin
Error := false;
  if NumIntervals <= 0 then
Error := true
else
begin
Spacing := (UpperLimit - LowerLimit) / (2 * NumIntervals);
Point := LowerLimit;
OddSum := 0;
EvenSum := 0;

Interval := 1;
while Interval <= (2 * NumIntervals - 1) do
begin
Point := Point + Spacing;
Func(Point, Y);
if (Interval mod 2) <> 0 then
OddSum := OddSum + Y

```



```
else
EvenSum := EvenSum + Y;

Interval := Interval + 1;
end;

Func(UpperLimit, YUpper);
Func(LowerLimit, YLower);
LimitsValue := YUpper + YLower;
Integral := Spacing * (LimitsValue + 2 * EvenSum + 4 * OddSum) / 3;
end;
end;
const
LowerLimit = 0.;
UpperLimit = 2.;
NumIntervals = 10;
var
Integral : real;
Error : boolean;
i : integer;
begin
i := 0;
while i < 100 do
begin
Simpson(LowerLimit, UpperLimit, NumIntervals, Integral, Error);
i := i + 1;
end;
end.
end.
```

Figura A.4: O Programa *Simpson*

```

Program Arvore;
  { programa para gerar uma arvore binaria ordenada em um vetor }
Var
  param, ultimovago,raiz,vago : integer;
var
  FilhoDir, FilhoEsq : array [1..200] of integer;
Var
  VetorValor : array [1..200] of integer;
Var
  Achou : boolean;

Procedure Insere ( Var p : Integer);
Var no,ant : integer;
begin
  if vago <= UltimoVago Then
    begin
      Achou := False;
      ant := 0;
      No := Raiz;
      while ( No <> 0 ) and (achou = false ) do
        If vetorValor[No] = Param Then begin
          achou := true;
          end
        else
          begin
            ant := no;
            if VetorValor[no] > Param Then No := FilhoEsq[No]
            Else No := filhoDir[No];
          end;
      If not achou Then
        begin
          No := Vago;
          Vago := Vago + 1;
          FilhoEsq[No] := 0;
          FilhoDir[No] := 0;
          VetorValor[No] := Param;
        end;
    end;
end;

```

```
    If Ant = 0 Then
        Raiz := No
    else
        begin
            If param < VetorValor[Ant] Then
                FilhoEsq[Ant] := No
            Else FilhoDir[Ant] := No;
        end;
    end;
end;
end;
begin
    raiz := 0;
    vago := 1;
    ultimoVago := 200;
    param := 5;
    insere(param);
    param := 7;
    insere (param);
    param := 20 ;
    Insere (param);
    Param := 15;
    insere(param);
end.
```

Figura A.5: O Programa *Árvore*

Apêndice B

A Utilização das Unidades Funcionais

Ainda que valiosas as conclusões tiradas na Subseção 5.3.2, a partir de observação dos tempos de execução dos programas teste, para o desenvolvimento de um processador Super Escalar eficiente, existe ainda um outro aspecto que merece ser examinado.

Esse aspecto refere-se ao nível de ocupação, durante a execução dos programas de teste, das unidades funcionais do modelo de arquitetura proposto.

EXECUÇÃO SEQUENCIAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	17800	24,45
2 ALUs	0	0,00
1 FPU	0	0,00
1 MEM	49800	68,43
1 BRANCH	5172	7,10

Total de IMFs Executadas: 72772

Tabela B.1: UFs Ativas do Programa *BCD* – Execução Seqüencial

Nas Tabelas B.1, B.2, B.3 e B.4 mostramos a taxa de ocupação das diversas unidades funcionais do modelo, para cada uma das configurações, quando da execução do programa *BCD*. Observamos que temos apenas a tabela da Configuração 1 para a execução seqüencial. Isso é possível pois seja qual for a configuração, obrigatoriamente, apenas uma unidade funcional de cada tipo é usada enquanto as demais

permanecem ociosas, portanto a única tabela é suficiente para nossa avaliação.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	13792	18,89
2 ALUs	2004	2,74
1 FPU	0	0,00
1 MEM	49800	68,23
1 BRANCH	3299	4,52

Total de IMFs Executadas: 72983

Tabela B.2: UFs Ativas do Programa *BCD* – Configuração 1

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	8242	16,38
2 ALUs	3100	6,16
3 ALUs	1050	2,08
4 ALUs	52	0,10
1 FPU	0	0,00
2 FPUs	0	0,00
1 MEM	24744	49,20
2 MEMs	12528	24,91
1 BRANCH	3299	6,56

Total de IMFs Executadas: 50921

Tabela B.3: UFs Ativas do Programa *BCD* – Configuração 2

Nas tabelas, a coluna Nº de IMFs refere-se ao número de de instruções longas executadas com uma ou mais unidades funcionais ativas simultaneamente. Por exemplo na Tabela B.2 referente a execução condicional do programa *BCD* na Configuração 1, vemos que durante a execução de 13792 instruções longas uma ALU permaneceu ativa, e durante a execução 2004 instruções longas as duas ALUs presentes na arquitetura operaram concorrentemente. Conseqüentemente durante 57187 ciclos de máquina nenhuma ALU estava sendo usada, já que o total de IMFs executadas é de 72983. Isto é, $57187 = 72983 - (13792 + 2004)$.

Ainda em relação a Tabela B.2, a coluna % do Tempo de Execução mostra que

uma ALU permaneceu ocupada por 18,89%, e duas ALUs permaneceram ocupadas por 2,74% do tempo de execução. Conseqüentemente durante 78,35% do tempo de execução nenhuma das ALUs presentes na configuração, foi ativada.

Examinando as Tabelas B.1, B.2, B.3 e B.4, percebemos que qualquer que seja a configuração, a nenhuma FPU é usada. Isso ocorre pois, o programa BCD não manipula dados em ponto-flutuante. Portanto é útil determo-nos apenas no exame da utilização das demais unidades funcionais presentes nas três configurações do CONDEX testadas.

Observando as Tabelas B.2 e B.3, correspondente a execução condicional do programa *BCD* na Configuração 1 e na Configuração 2 respectivamente, vemos quando dobramos o número de ALUs e MEMs (Configuração 2), as instruções do programa apresentavam paralelismo de baixo-nível suficiente, para que até quatro ALUs fossem usadas simultaneamente (embora em apenas 0,1% do tempo total de execução). Da mesma forma vemos que na Configuração 1, a presença de uma única MEM, não possibilita que o paralelismo presente no programa seja totalmente explorado, permanecendo fortemente ocupada 68,23% do tempo de execução. Essa conclusão é ratificada quando observamos a taxa de ocupação das MEMs presentes na Configuração 2, ambas permanecem ativas 24,91% e apenas uma 49,20% do tempo de execução. Já o percentual de uso da unidade de desvios (BRANCH) na Configuração 2 é maior, embora o número de instruções de desvio executadas seja o mesmo que o da Configuração 1, isso ocorre pois o tempo total de execução total na Configuração 2 é menor que na Configuração 1.

Se observarmos agora a Tabela B.4, verificamos as quatro ALUs adicionais presentes na Configuração 3, permanecem não utilizadas por quase toda a execução condicional do programa teste. Existe somente uma instrução longa na qual foi possível utilizar sete ALUs concorrentemente. Ou seja, o investimento correspondente ao acréscimo de quatro ALUs não resultou em um tempo menor de execução. Já adição das duas MEMs reverteu na redução de ciclos de máquina necessários para a execução do programa *BCD*. Podemos notar que embora as dependências de dados houvessem impedido que a quarta MEM fosse usada por uma parcela significativa do tempo de execução, durante 711 ciclos de máquina as quatro MEMs operaram simultaneamente. Podemos visualizar melhor o uso das MEMs durante a interpretação do programa *BCD* na Configuração 3, no histograma da Figura B.1.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	8014	21,47
2 ALU	3024	8,10
3 ALU	1177	3,15
4 ALU	50	0,13
5 ALU	0	0,00
6 ALU	0	0,00
7 ALU	1	0,00
8 ALU	0	0,00
1 FPU	0	0,00
2 FPU	0	0,00
3 FPU	0	0,00
4 FPU	0	0,00
1 MEM	12796	34,29
2 MEM	10774	28,87
3 MEM	4204	11,26
4 MEM	711	1,90
1 BRANCH	3299	8,84

Total de IMFs Executadas: 37315

Tabela B.4: UFs Ativas do Programa *BCD* – Configuração 3

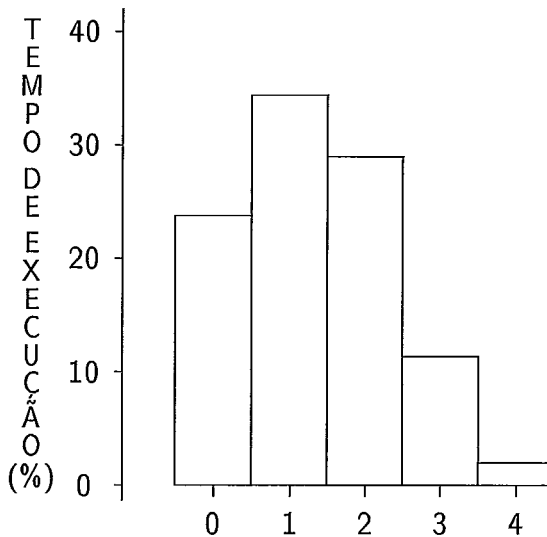


Figura B.1: UFs do tipo MEM Concorrentemente Ativas – Configuração 3

Na Figura B.1 vemos que somente durante 23,66% do tempo de execução nenhuma MEM está ativa, 34,29% uma MEM está ativa, 28,87% duas MEMs operam

concorrentemente, 11,26% três MEMs são usadas em paralelo e 1,90% do tempo as quatro MEMs presentes no modelo de arquitetura correspondente a Configuração 3, estão em operação simultaneamente.

Destoando do histograma da Figura B.1 que mostra como as MEMs são usadas durante a execução do programa teste, temos na Figura B.2 o histograma que mostra o uso das oito ALUs presentes na Configuração 3.

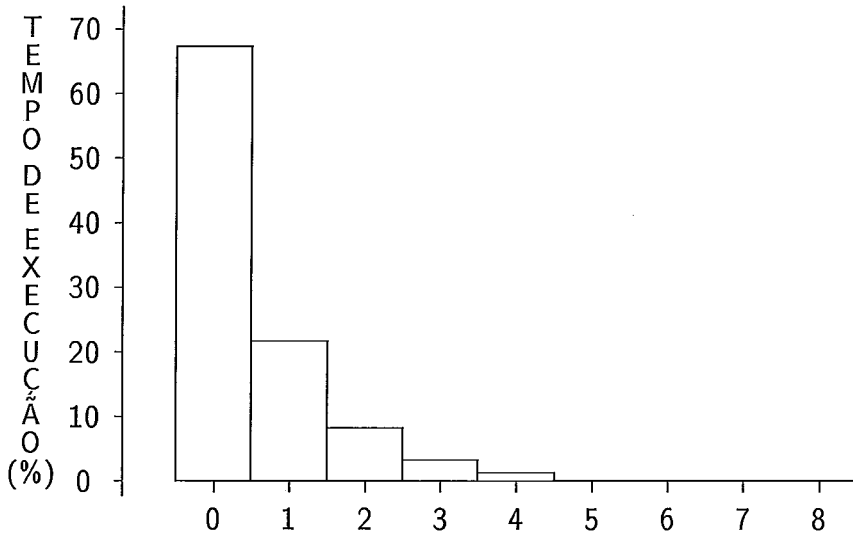


Figura B.2: UFs do tipo ALU Concorrentemente Ativas – Configuração 3

Na Figura B.2 percebemos que 67,12% do tempo de execução, nenhuma das ALUs presentes na Configuração 3 está em operação. Uma única ALU opera por 21,47% do tempo, duas ALUs operam juntas por 8,10%, três por 3,15% e quatro ALUs operam concorrentemente por apenas 0,13% do tempo de execução do programa *BCD*. As outras quatro ALUs da arquitetura, permanecem praticamente ociosas. A única exceção é quando sete ALUs estão ativas em uma instrução.

Examinando os dois histogramas (Figuras B.1 e B.2), concluímos que o uso das unidades funcionais MEM para esse programa teste, para a Configuração 3, é mais eficiente que o uso das ALUs. Provavelmente para o programa *BCD*, uma configuração alternativa onde tivéssemos quatro MEMs e quatro ALUs (além da unidade funcional *BRANCH*) resultaria num tempo de execução próximo ao obtido na configuração examinada.

Quanto a única unidade funcional *BRANCH* presente na Configuração 3 (Tabela B.4), verificamos que esta executou o mesmo número de instruções que a da

Configuração 2 (Tabela B.3), porém o seu percentual de utilização subiu de 6,56% para 8,84%, já que o número de ciclos necessários para a execução do programa, caiu de 50921 para 37315.

Nas Figuras B.3, B.4, B.5 e B.6 vemos os histogramas que mostram percentual de tempo que cada unidade funcional permaneceu ocupada durante a interpretação do programa *BCD*, para as configurações testadas.

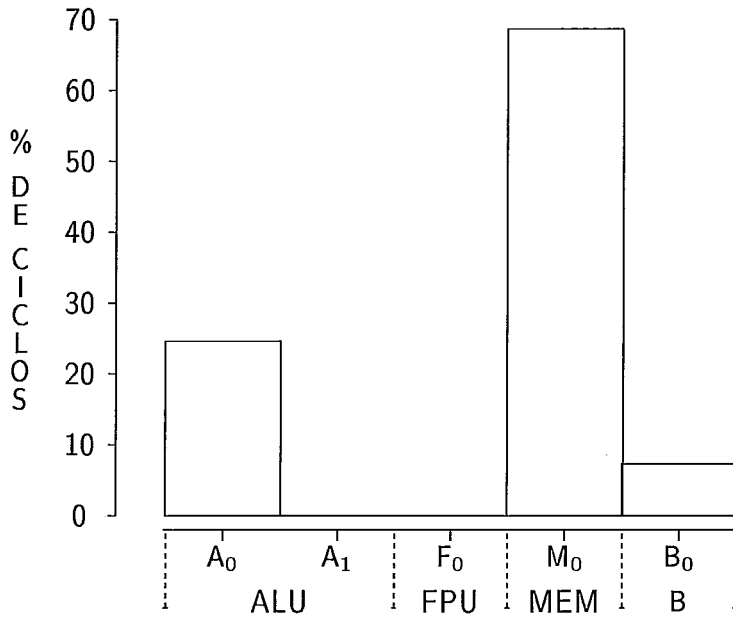


Figura B.3: UFs Ativas na Configuração 1 Durante a Execução Sequencial do Programa *BCD*

Apresentamos apenas um histograma (Figura B.3), relativo a execução sequencial, já que qualquer que seja a configuração testada, apenas uma unidade funcional permanece ativa em cada ciclo de processador. Quando da execução sequencial a primeira ALU permanece ocupada durante 24,45%, a primeira MEM é utilizada por 68,43% e a única unidade de desvio (BRANCH) opera por 7,10%. As demais unidades funcionais são mantidas ociosas por todo o tempo da interpretação do programa *BCD*.

Para a execução condicional na Configuração 1 (Figura B.4), temos que a primeira ALU permanece ativa por durante 21,63% do tempo de execução, ao passo que a segunda por somente 2,74%. A única FPU não é utilizada, a única MEM é mantida ocupada 68,23% e a unidade de desvio da configuração fica ocupada por 4,52% do tempo de execução.

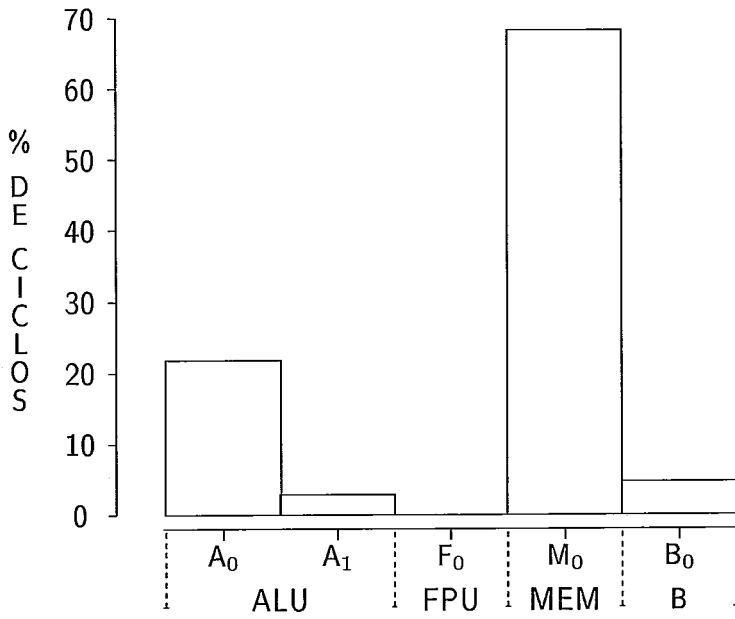


Figura B.4: UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa *BCD*

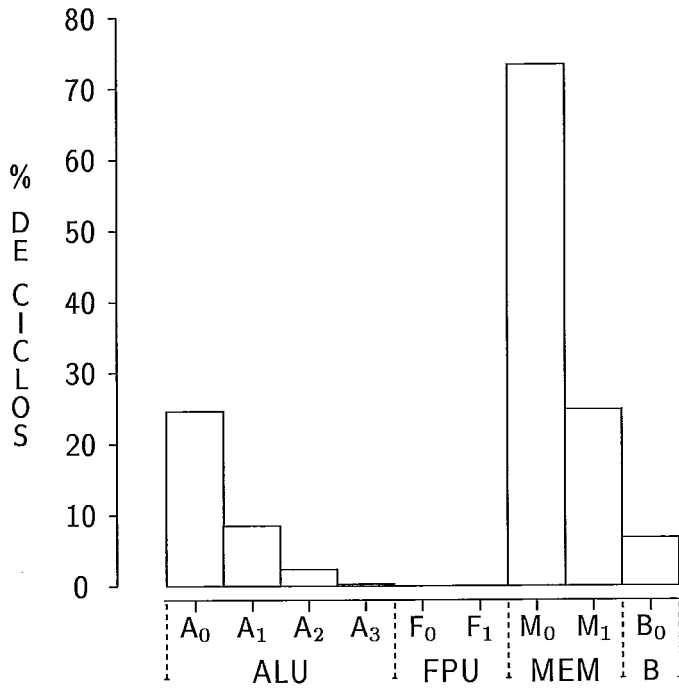


Figura B.5: UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa *BCD*

Podemos observar os efeitos da execução condicional na Configuração 2 no histograma mostrado na Figura B.5. Nesse caso a primeira ALU é usada 24,43%, a segunda 8,26%, a terceira 2,18% e a quarta por apenas 0,10% dos ciclos de processador necessários para a execução do Programa *BCD*. As duas FPUs não são ativadas nesse teste, as duas MEMs permanecem processando por 74,11% e 24,91% respectivamente. Finalmente a unidade *BRANCH* é usada por 6,56% do tempo.

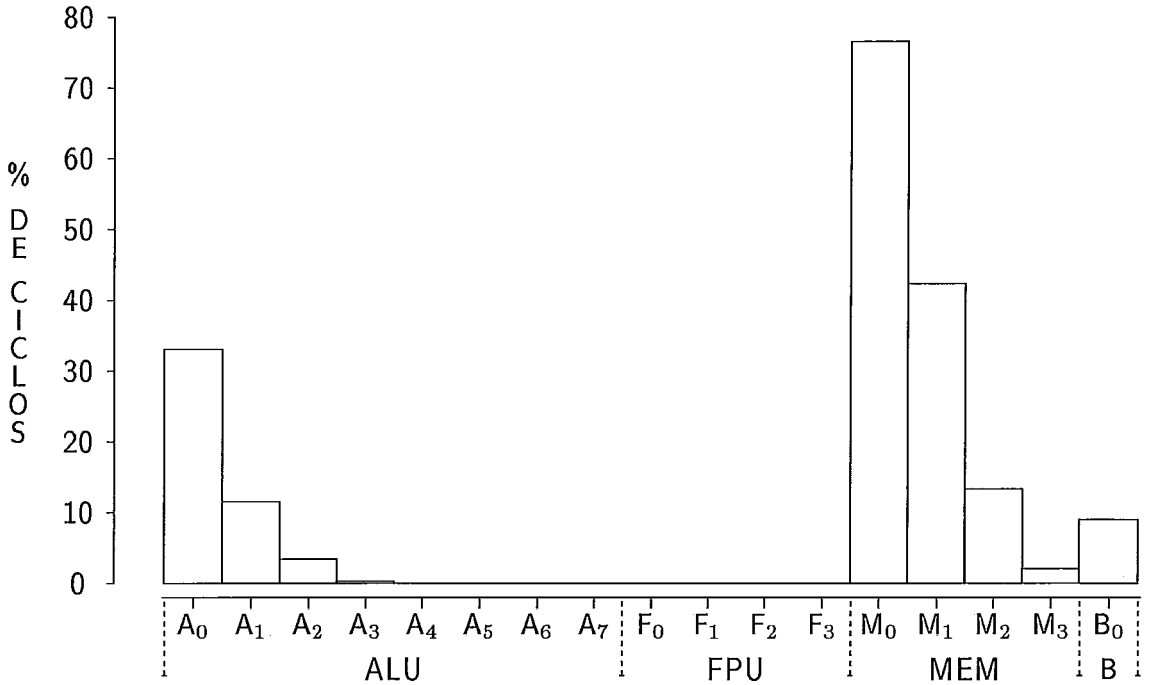


Figura B.6: UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa *BCD*

Para a Configuração 3, cujo histograma ilustrativo da atividade das unidades funcionais, durante a execução condicional, é mostrado na Figura B.6, vemos que as quatro primeiras ALUs permanecem utilizadas respectivamente por 32,85%, 11,38%, 3,28% e 0,13%. As quatro ALUs restantes não são utilizadas durante a execução, bem como as quatro FPUs presentes na configuração. As MEMs são ativadas por 76,32%, 42,03%, 13,16% e 1,90% dos ciclos de processador. Por fim a unidade de desvio é mantida ocupada por 8,84% do tempo total necessário para a execução do programa teste.

Nas tabelas das Tabelas B.5, B.6, B.7 e B.8 mostramos a taxa de ocupação de cada uma das unidades funcionais do modelo, para cada uma das configurações, quando da execução do programa Branch. Como no caso do programa *BCD* exa-

minado anteriormente, apresentamos apenas uma tabela para a execução seqüencial cujos valores podem ser utilizados para as três configurações examinadas.

EXECUÇÃO SEQUENCIAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	28171	21,56
2 ALUs	0	0,00
1 FPU	0	0,00
1 MEM	92188	70,58
1 BRANCH	10244	7,84

Total de IMFs Executadas: 130603

Tabela B.5: UFs Ativas do Programa *Branch* – Execução Seqüencial

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	23045	20,45
2 ALUs	2563	2,27
1 FPU	0	0,00
1 MEM	92188	81,81
1 BRANCH	5124	4,54

Total de IMFs Executadas: 112675

Tabela B.6: UFs Ativas do Programa *Branch* – Configuração 1

Examinando as Tabelas B.5 e B.6, percebemos que o programa *Branch* possui um grande número de instruções de transferência com a memória, já que em 70,58% do tempo de execução a única unidade MEM da Configuração 1 é usada no caso da execução seqüencial. Durante a execução condicional do programa teste, essa utilização aumenta para 81,81%. Esse aumento do percentual de utilização deve-se a redução do número total de instruções longas executadas, causada não somente pelo paralelismo presente nas instruções do programa, como também pela eliminação de certas instruções de desvio, desnecessárias quando da compactação condicional. Facilmente podemos constatar nas tabelas, que o número de instruções de desvio executadas caiu de 10244 na execução seqüencial, para 5124 instruções na execução

condicional.

Quando dobramos os recursos do modelo, isto é, passamos da Configuração 1 para Configuração 2 (Tabela B.7), percebemos que a inclusão de duas novas ALUs não reverteu em benefício para o tempo total de execução no caso da compactação condicional. Isso é facilmente constatável, já que apenas uma instrução longa resultou em três ALUs operando concorrentemente, e uma segunda causou a ativação de quatro ALUs no mesmo ciclo de máquina. Mesmo duas ALUs operaram simultaneamente apenas 2,85% do tempo de execução, o que corresponde a 2560 IMFs de um total de 89627 executadas.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	23044	25,71
2 ALUs	2560	2,85
3 ALUs	1	0,00
4 ALUs	1	0,00
1 FPU	0	0,00
2 FPUs	0	0,00
1 MEM	56332	62,85
2 MEMs	17928	20,00
1 BRANCH	5124	5,71

Total de IMFs Executadas: 89627

Tabela B.7: UFs Ativas do Programa *Branch* – Configuração 2

Ainda com respeito a Configuração 2 (Tabela B.7), observamos que o acréscimo de uma MEM é benéfico à aceleração da execução do programa *Branch*. Notamos que as duas MEMs ficam ativas por parcelas significativas do tempo de execução, permanecendo ambas ociosas durante 15367 ciclos de máquina.

Examinando as Tabelas B.5, B.6, B.7 e B.8, concluímos que qualquer que seja a configuração, nenhuma FPU é ativada. Isso ocorre pois, o programa *Branch* a exemplo do programa *BCD* não manipula dados em ponto-flutuante.

Quando evoluímos da Configuração 2 para a Configuração 3 ratificamos a observação feita anteriormente de que o programa *Branch* não possui paralelismo de baixo-nível capaz de manter ocupadas mais de duas ALUs simultaneamente.

Essa constatação é comprovada quando examinamos Tabela B.8, correspondente a Configuração 3. Claramente seis das oito ALUs presentes na configuração, nunca são usadas.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	17924	20,58
2 ALU	5121	5,88
3 ALU	0	0,00
4 ALU	0	0,00
5 ALU	1	0,00
6 ALU	0	0,00
7 ALU	0	0,00
8 ALU	0	0,00
1 FPU	0	0,00
2 FPU	0	0,00
3 FPU	0	0,00
4 FPU	0	0,00
1 MEM	61452	70,58
2 MEM	0	0,00
3 MEM	10240	11,76
4 MEM	4	0,00
1 BRANCH	5124	5,88

Total de IMFs Executadas: 87063

Tabela B.8: UFs Ativas do Programa *Branch* – Configuração 3

O mais interessante da Tabela B.8 é a atividade das MEMs presentes na configuração: uma única MEM é ativada em 70,58% do tempo de execução, e três MEMs são ativadas em 11,76% desse mesmo tempo. O curioso é que nenhuma instrução longa mantém duas ou quatro MEMs operando concorrentemente. Concluímos nesse caso que o aumento de uma MEM na Configuração 2 já seria suficiente para atingir o tempo de execução verificado na Configuração 3, para esse programa em particular.

O percentual de uso da unidade de desvios (BRANCH) na Configuração 3 é maior, embora o número de instruções de desvio executadas seja o mesmo que o da Configuração 1 (isto é 5124 instruções), pois o tempo total de execução total na Configuração 3 é menor que na Configuração 2.

É interessante observar como o acréscimo de unidades MEMs influencia no tempo de execução, no caso da execução condicional. Para isso propusemos uma nova configuração, a Configuração 4 cujos recursos são o dobro da Configuração 3, e uma Configuração 5 cujos recursos são o dobro da Configuração 4, ou seja a Configuração 4 possui oito MEMs e a Configuração 5 dezesseis MEMs. O gráfico na Figura B.7, ilustra essa situação para o programa Branch. Ou seja, variamos o número de MEMs de 1 até 16 (segundo potências de 2).

Podemos verificar na Figura B.7 que a configuração com quatro unidades MEMs, é que oferece o melhor desempenho. Para as configurações com oito e dezesseis MEMs, o desempenho não aumenta. Contudo quando comparamos o desempenho da configuração com duas MEMs com a que possui quatro MEMs, verificamos que a redução do número de ciclos de máquina necessários para a execução do programa teste, foi de apenas 2,86%. Desse modo, considerando agora critérios de custo, podemos concluir que a inclusão de duas unidades funcionais do tipo MEM, é bastante razoável sob o ponto de vista “custo \times desempenho.” Por outro lado é correto dizer que uma maior queda na tempo de execução, nesse caso, não depende do número de recursos disponíveis, mas sim das dependências de dados presentes no programa.

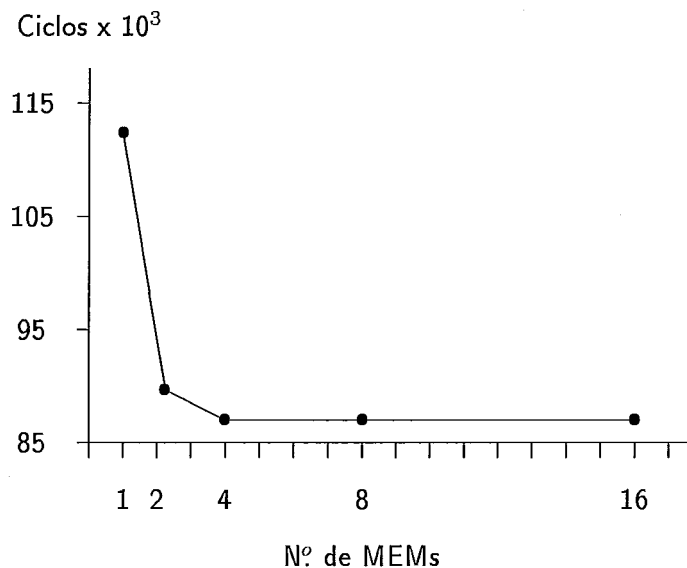


Figura B.7: Efeito das Unidades MEMs no Desempenho – *Branch*

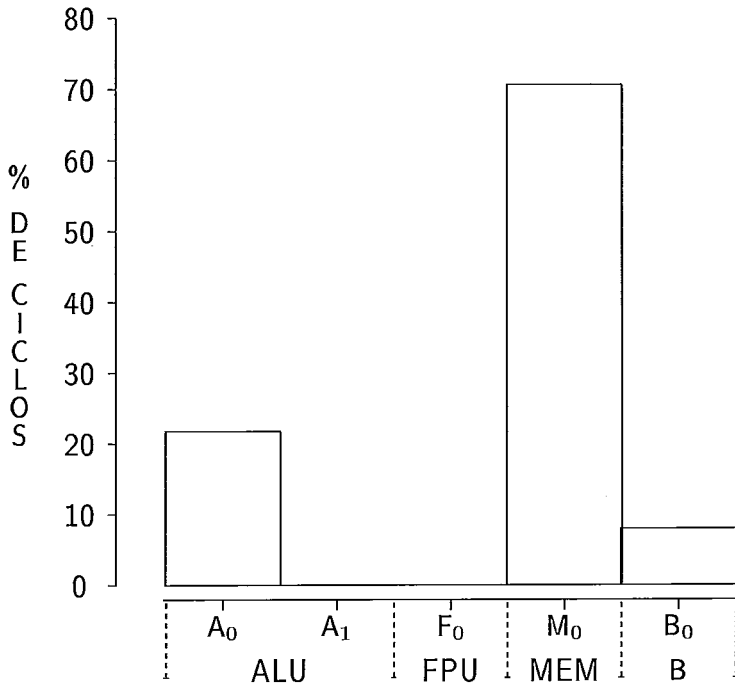


Figura B.8: UFs Ativas na Configuração 1 Durante a Execução Sequencial do Programa *Branch*

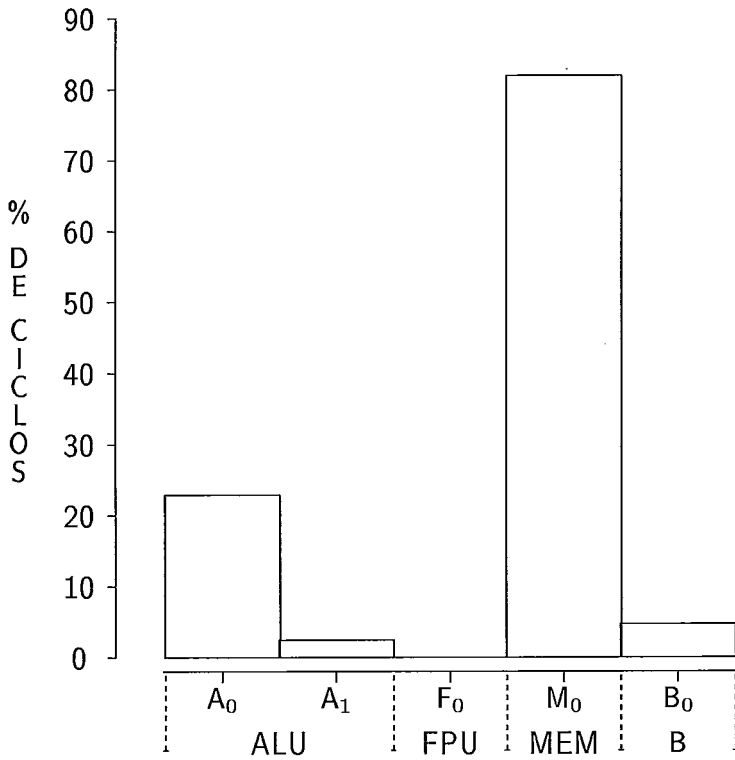


Figura B.9: UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa *Branch*

Temos um panorama das unidades funcionais ativas durante a execução do programa *Branch* nos histogramas mostrados nas Figuras B.8, B.9, B.10 e B.11. Para execução seqüencial temos um único histograma (Figura B.8), visto que não independentemente da configuração, apenas uma unidade funcional é usada em cada ciclo de processador.

Na Figura B.8 vemos que uma ALU permanece em uso por 21,56% dos ciclos de processador requeridos, uma MEM por 70,58% e a unidade de desvio por 7,84% do tempo de execução do programa *Branch*. As FPU's não são usadas.

Quando da execução condicional na Configuração 1 (Figura B.9), vemos que a primeira ALU permanece executando por 22,72% dos ciclos, ao passo que a segunda por apenas 2,27%. A unidade de memória MEM fica ocupada 81,81% e a BRANCH por 4,54% do tempo de execução do programa *Branch*. Unidades para aritmética em ponto flutuante não são requeridas.

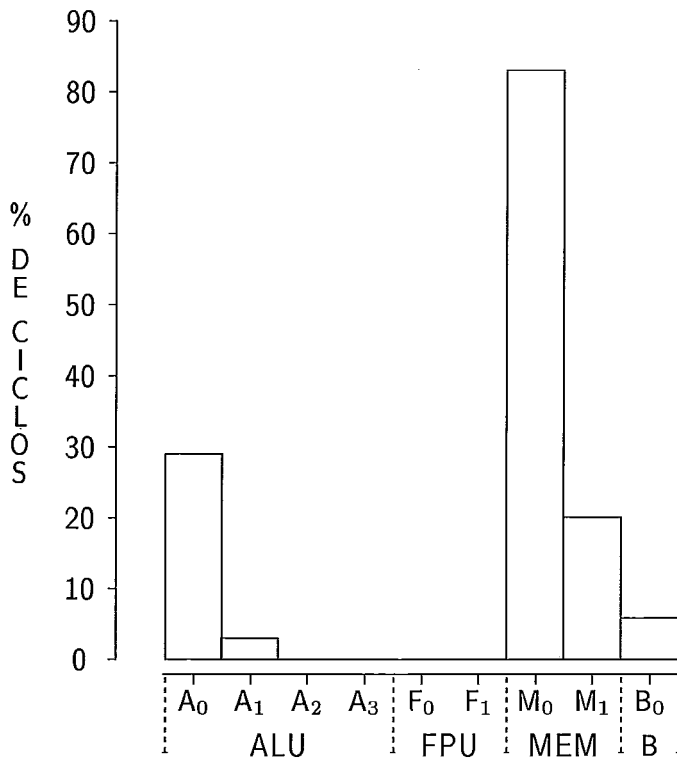


Figura B.10: UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa *Branch*

No histograma da Figura B.10 vemos que a execução condicional na Configuração 2 mantém ocupadas respectivamente, a primeira e segunda ALU por 28,56%

e 2,85% dos ciclos de processador. As duas outras ALUs não são usadas. As duas unidades de acesso à memória permanecem em uso por 82,85% e 20,00%, e a unidade de processamento de desvios por 5,71% do tempo de execução. As unidades para aritmética em ponto flutuante não são usadas durante o teste.

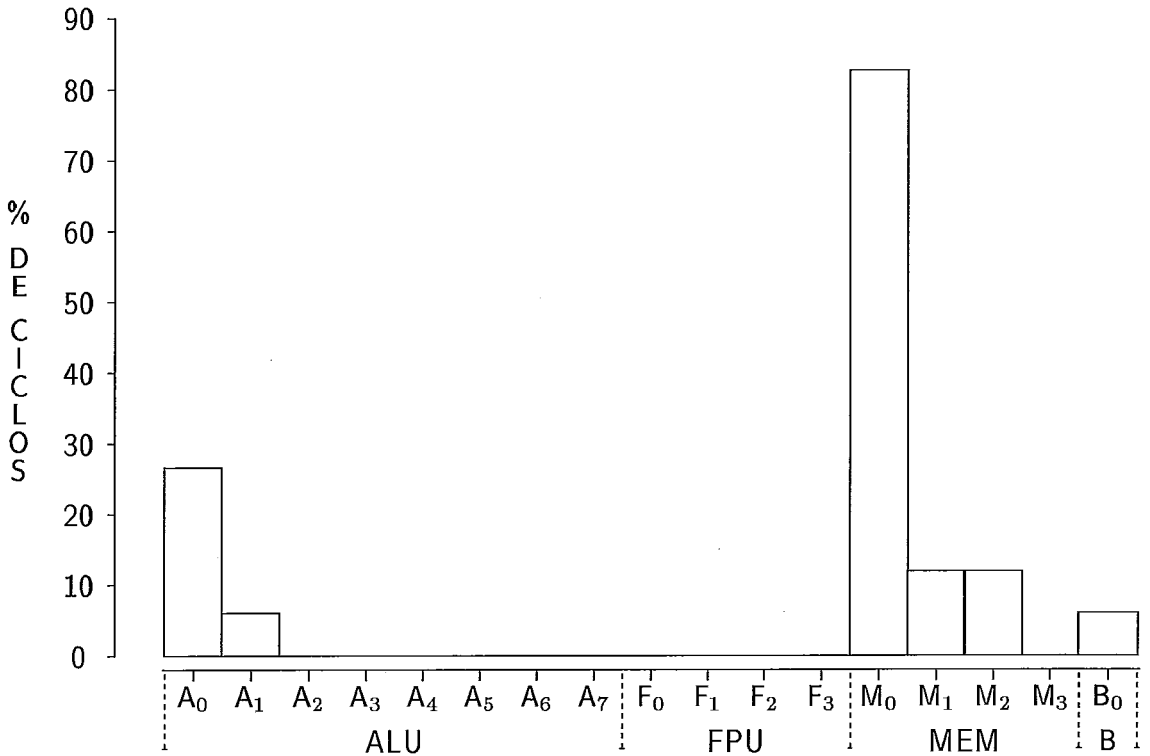


Figura B.11: UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa *Branch*

Na Configuração 3 do modelo CONDEX (Figura B.11), as duas primeiras ALUs processam por 26,46% e 5,88% do tempo de execução. As seis ALUs restantes e as quatro FPUs da configuração não são usadas. Três das unidades de acesso à memória (MEMs), permanecem ocupadas por 82,34%, 11,76% e 11,76% dos ciclos de processador dispendidos durante a execução. A quarta MEM não é ativada e a unidade de processamento de desvios BRANCH, opera por 5,88% do tempo de execução.

Nas Tabelas B.9, B.10, B.11 e B.12 mostramos a taxa de ocupação de cada uma das unidades funcionais do CONDEX para cada uma das configurações, quando da execução do programa Simpson.

De modo diferente dos dois programas examinados anteriormente, no programa *Simpson*, temos a presença de instruções que manipulam dados em ponto-flutuante. Na Tabela B.9 correspondente a execução seqüencial do programa testado na Configuração 1, temos que 269828 ciclos de máquina são necessários para tanto. Desse total 19,68% são gastos em operações em ALU, 8,63% em operações em FPU, 4,67% na unidade BRANCH, e a grande parte do tempo, isto é 67,01% é dispendida em instruções de acesso à memória pela MEM.

EXECUÇÃO SEQUENCIAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	53108	19,68
2 ALUs	0	0,00
1 FPU	23300	8,63
1 MEM	180816	67,01
1 BRANCH	12604	4,67

Total de IMFs Executadas: 269828

Tabela B.9: UFs Ativas do Programa *Simpson* – Execução Seqüencial

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	27306	12,09
2 ALUs	12901	5,71
1 FPU	23300	10,32
1 MEM	180816	80,10
1 BRANCH	9604	4,25

Total de IMFs Executadas: 225723

Tabela B.10: UFs Ativas do Programa *Simpson* – Configuração 1

Quando nos reportamos à Tabela B.10 relativa a execução condicional verificamos que o tempo de execução cai de 269828 ciclos da execução seqüencial para 225723 ciclos. É portanto uma redução de 16.64%. Embora as duas ALUs sejam usadas concorrentemente em apenas 5.71% do tempo, a extração do paralelismo presente no código em do programa teste, proporcionou essa redução no tempo de

execução.

É evidente que quando executamos o programa *Simpson* na Configuração 2 (Tabela B.11), a redução no tempo de execução, se comparada com a execução sequencial, cresce expressivamente: é de 40,10%. Esse fato nos leva a concluir que a inclusão de recursos adicionais na arquitetura é útil nesse caso específico. Observamos que o total das instruções com inteiros, presentes no programa testado, foram distribuídas entre as quatro ALUs presentes na configuração, embora somente em 300 instruções longas todas se mantivessem ativas simultaneamente. Isso corresponde a apenas 0,18% do número de ciclos de máquina, necessários para a interpretação. A ocupação concorrente das duas FPUs também foi pobre: foi por apenas 0,43% do tempo total de execução. Uma única ALU esteve ocupada por 13,55% do tempo e nenhuma ALU foi usada por 86,01% dos ciclos de máquina requeridos para a execução do programa. Uma melhor idéia do uso das FPUs existentes na Configuração 2 pode ser vista no histograma contido na Figura B.12.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	25204	15,59
2 ALUs	10202	6,31
3 ALUs	2100	1,29
4 ALUs	300	0,18
1 FPU	21900	13,55
2 FPUs	700	0,43
1 MEM	57010	35,27
2 MEMs	61903	38,30
1 BRANCH	8604	5,32

Total de IMFs Executadas: 161620

Tabela B.11: UFs Ativas do Programa *Simpson* – Configuração 2

Voltando a Tabela B.11, notamos que as instruções de transferência com a memória, mantiveram ocupadas concorrentemente por 38,30% do tempo de execução, as duas MEMs da arquitetura e 35,27% do tempo, uma das MEMs, ficando as duas MEMs ociosas por 26,42% do tempo total de execução do programa *Simpson* na Configuração 2.

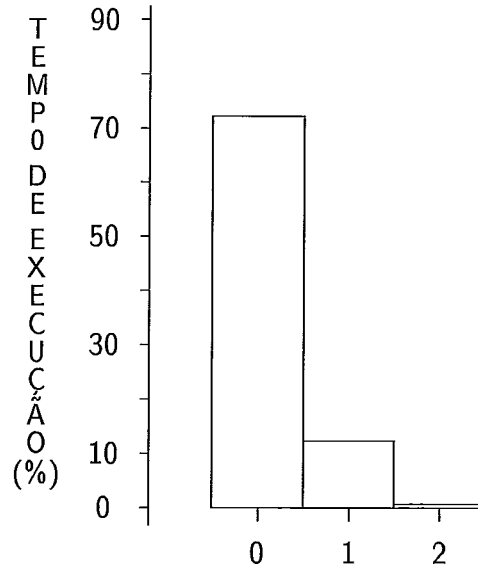


Figura B.12: UF's do tipo FPU Concorrentemente Ativas – Configuração 2

EXECUÇÃO CONDICIONAL		
Nº de UF's Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	25204	16,84
2 ALU	10202	6,81
3 ALU	2100	1,40
4 ALU	300	0,20
5 ALU	0	0,00
6 ALU	0	0,00
7 ALU	0	0,00
8 ALU	0	0,00
1 FPU	21700	14,50
2 FPU	800	0,53
3 FPU	0	0,00
4 FPU	0	0,00
1 MEM	55010	36,76
2 MEM	42203	28,20
3 MEM	600	0,40
4 MEM	9900	6,61
1 BRANCH	8604	5,75

Total de IMFs Executadas: 149620

Tabela B.12: UF's Ativas do Programa *Simpson* – Configuração 3

Na Tabela B.12 correspondente as unidades funcionais ativas na Configura-

ção 3, observamos que a inclusão das quatro novas ALUs não refletiu na redução do tempo de execução já que nunca mais de quatro ALUs estiveram em operação em paralelo. A mesma observação vale para as duas FPUs adicionais. Apenas as duas novas MEMs foram realmente úteis para que o tempo de execução na Configuração 3 fosse 7,21% menor que o da Configuração 2.

O percentual de tempo em que as unidades funcionais de cada configuração do CONDEX, permanecem processando durante a execução condicional do programa *Simpson* é mostrado nas Figuras B.13, B.14, B.15 e B.16. Do mesmo modo como nos programas examinados anteriormente, para execução seqüencial temos apenas um histograma, o da Figura B.13.

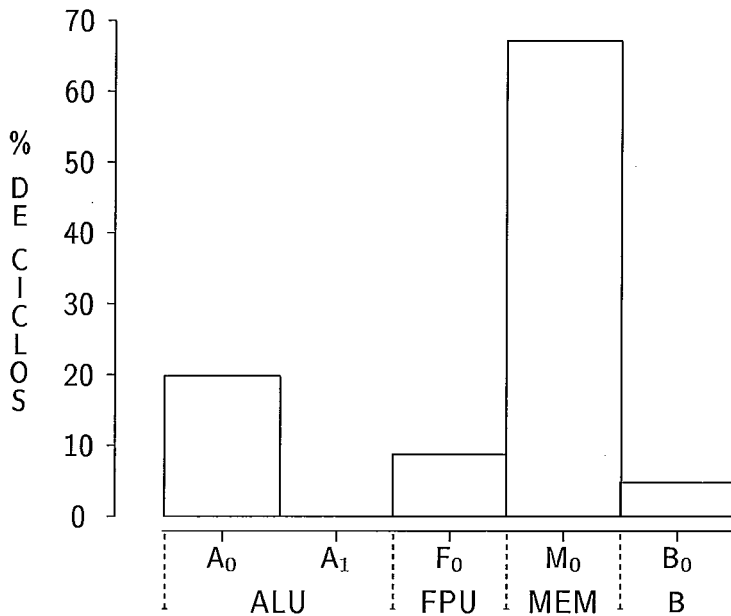


Figura B.13: UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa *Simpson*

Na Figura B.13 vemos que durante a execução seqüencial do programa *Simpson* uma ALU permanece ocupada por 19,68%, uma FPU por 8,67%, uma unidade de acesso a memória por 67% e a unidade BRANCH por 4,67% do tempo total de execução. As demais unidades funcionais permanecem inativas.

Durante a execução condicional na Configuração 1 do Condex, vemos na Figura B.14 que a primeira ALU é usada por 17,80%, e a segunda por 5,71% do tempo. As unidades FPU, MEM e BRANCH da arquitetura, permanecem ocupadas respectivamente, por 10,32%, 80,10% e 4,25% do total de ciclos de processador necessários

para a execução do programa.

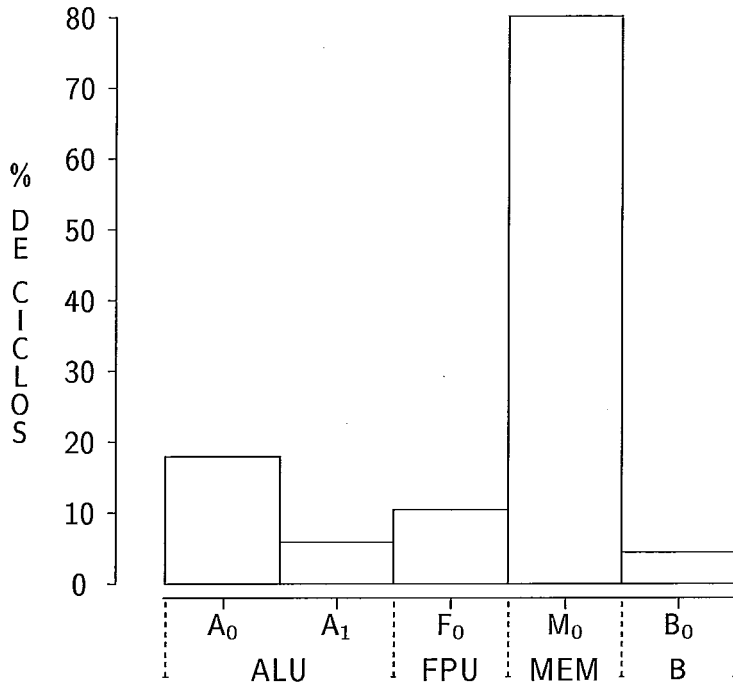


Figura B.14: UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa *Simpson*

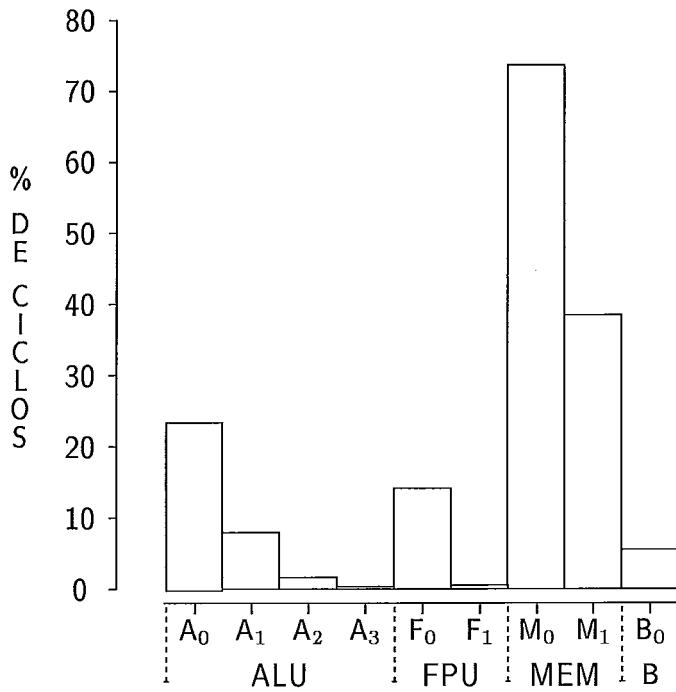


Figura B.15: UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa *Simpson*

Na Configuração 2 (Figura B.15), a execução condicional do programa *Simpson* mantém ocupadas as quatro ALUs por 23,37%, 7,78%, 1,47% e 0,18%; as duas FPU's por 14,03% e 0,43%; as duas unidades de acesso à memória por 73,57% e 38,30%; e a unidade de processamento de desvios por 5,32% do tempo total dispendido.

Finalmente na Figura B.16, temos o perfil de ocupação das unidades funcionais presentes na Configuração 3, quando da execução condicional do programa *Simpson*. Apenas quatro das oito ALUs são ativadas durante a execução, por: 25,25%, 8,41%, 1,60% e 0,20% do tempo total. Do mesmo modo somente duas das quatro FPU's são envolvidas no processamento, por 15,03% e 0,53% dos ciclos enquanto as outras duas permanecem ociosas. As quatro unidades MEM são usadas respectivamente por 71,97%, 35,21%, 7,01% e 6,61% dos ciclos, e por fim unidade de processamento de desvios opera 5,75% do tempo de execução.

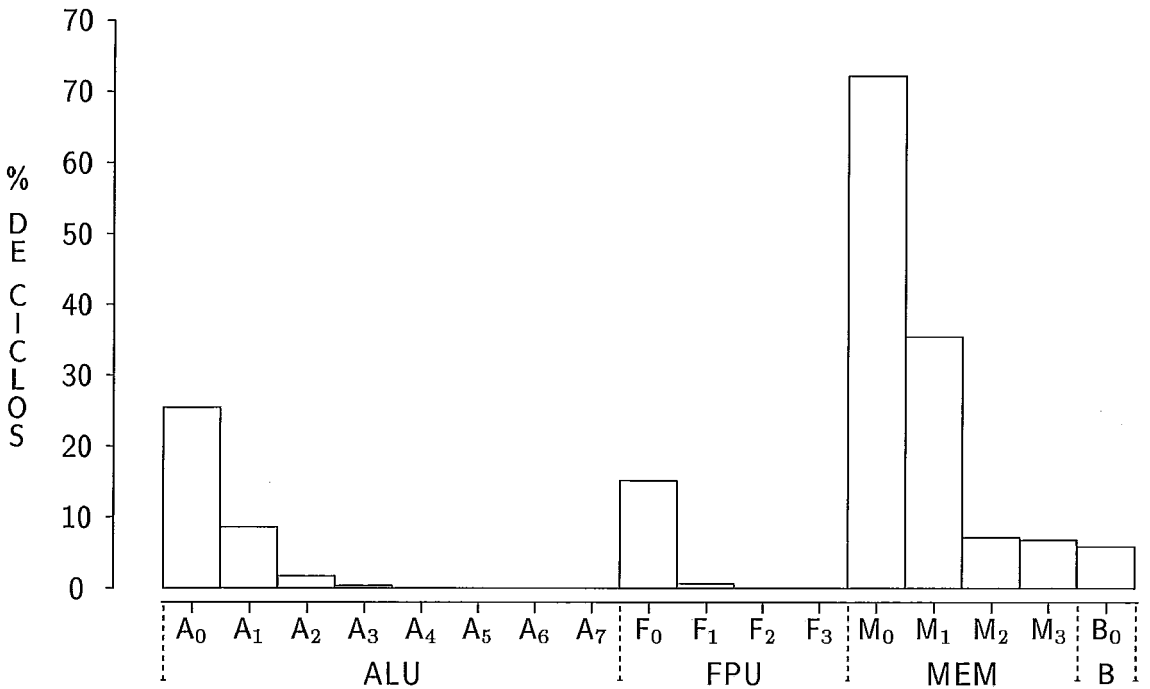


Figura B.16: UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa *Simpson*

Nas Tabelas B.13, B.14, B.15 e B.16 mostramos a taxa de ocupação de cada uma das unidades funcionais do CONDEX em cada uma das configurações testadas, quando da execução do programa *Livermore Loop número 24*. Do mesmo modo que nos programas vistos anteriormente, mostramos apenas uma tabela para a execução

seqüencial cujos valores podem ser utilizados para as três configurações examinadas.

Na Tabela B.13 correspondente a ocupação dos unidades funcionais execução seqüencial do programa *Livermore Loop número 24* na Configuração 1, percebemos que o programa não manipula dados em ponto-flutuante, e portanto a FPU presente na configuração nunca é necessária durante a sua execução. Temos o programa executada em 34390 ciclos de máquina. Desse total 34,65% são consumidos em instruções realizadas pela ALU, 57,18% em instruções executadas pela MEM e 8,15% pelas instruções manipuladas pela unidade BRANCH.

EXECUÇÃO SEQUENCIAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	11919	34,65
2 ALUs	0	0,00
1 FPU	0	0,00
1 MEM	19668	57,18
1 BRANCH	2803	8,15

Total de IMFs Executadas: 34390

Tabela B.13: UFs Ativas do Programa *Livermore Loop* – Execução Seqüencial

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	7705	22,41
2 ALUs	2107	6,13
1 FPU	0	0,00
1 MEM	19668	57,22
1 BRANCH	1404	4,08

Total de IMFs Executadas: 34367

Tabela B.14: UFs Ativas do Programa *Livermore Loop* – Configuração 1

Quando examinamos a Tabela B.14 relativa a atividade dos dispositivos funcionais durante a execução condicional na Configuração 1, verificamos que o tempo de execução é reduzido de 34390 ciclos de máquina correspondente a execução seqüencial, para 34367 ciclos (um redução de apenas 0,10%). Essa redução pouco

significativa deve-se, a uma anomalia da compactação condicional, de quando do escalonamento de instruções do “bs” em instruções longas do “bt.” Como os recursos são relativamente reduzidos, já que apenas duas ALUs estão presentes na configuração, e as demais unidades funcionais são únicas, o que ocorre é que na prática o “bs” acaba por ter mais instruções longas: algumas instruções são levadas para IMFs do “bt,” porém nenhuma (ou poucas) IMFs do “bs” são eliminadas.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	3504	17,22
2 ALUs	2800	13,76
3 ALUs	1	0,00
4 ALUs	703	3,45
1 FPU	0	0,00
2 FPUs	0	0,00
1 MEM	8406	41,32
2 MEMs	5631	27,68
1 BRANCH	1404	6,90

Total de IMFs Executadas: 20343

Tabela B.15: UFs Ativas do Programa *Livermore Loop* – Configuração 2

Quando dobramos os recursos do modelo, isto é, passamos da Configuração 1 para Configuração 2 percebemos, na Tabela B.15, que a inclusão desses novos recursos eliminou a anomalia presente na Configuração 1, e o número de ciclos de máquina necessários para a execução do programa cai de 34367 para 20343. Portanto uma redução no tempo de 40,80%.

Na Tabela B.15 vemos a inclusão de duas novas ALUs foi parcialmente responsável na aceleração obtida, pois quatro ALUs foram simultaneamente ativadas em 3,45% do tempo. Embora somente em ciclo de máquina tivéssemos três ALUs operando concorrentemente. A grande responsável pela aceleração, foi sem dúvida, a inclusão de uma nova unidade MEM. Nesse caso duas MEMs foram mantidas ativas por 27,68% e uma por 41,32% do tempo total de execução. A utilização das MEMs, da Configuração 2, quando da execução condicional do programa *Livermore*, pode ser vista no histograma da Figura B.17.

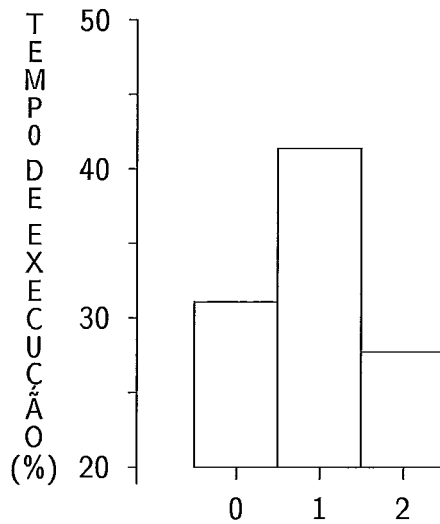


Figura B.17: UFs do tipo MEM Concorrentemente Ativas–Configuração 2

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	3504	17,23
2 ALU	2800	13,77
3 ALU	0	0,00
4 ALU	700	3,44
5 ALU	0	0,00
6 ALU	0	0,00
7 ALU	1	0,00
8 ALU	1	0,00
1 FPU	0	0,00
2 FPU	0	0,00
3 FPU	0	0,00
4 FPU	0	0,00
1 MEM	8404	41,33
2 MEM	5609	27,58
3 MEM	2	0,00
4 MEM	10	0,04
1 BRANCH	1404	6,90

Total de IMFs Executadas: 20311

Tabela B.16: UFs Ativas do Programa *Livermore Loop* – Configuração 3

Na Tabela B.16 relativa a Configuração 3, vemos que a presença de quatro novas ALUs e duas novas MEMs reduziu o tempo de execução do programa *Livermore*

de 20343 (Configuração 2), para 20311. Ou seja de apenas 32 ciclos de máquina. As novas unidades funcionais presentes na Configuração 3, permaneceram por quase todo o tempo inativas.

Nas Figuras B.18, B.19, B.20 e B.21 mostramos os histogramas que mostram o percentual de tempo em que as unidades funcionais permanecem operando, durante a execução do programa *Livermore Loop número 24*. Apenas um histograma, o da Figura B.18, é apresentado para a execução seqüencial, pois como dissemos anteriormente, nesse tipo de processamento, qualquer que seja a configuração apenas uma unidade funcional permanece ativa em cada ciclo de máquina.

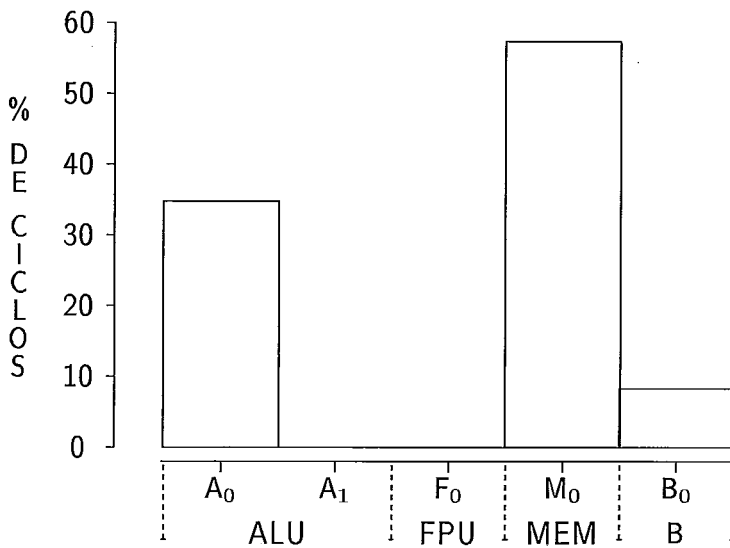


Figura B.18: UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa *Livermore Loop*

Na Figura B.18 relativa a execução seqüencial, vemos que uma ALU, uma unidade MEM e a unidade de processamento de desvios, permanecem em atividade respectivamente, por 34,65%, 57,18% e 8,15% do tempo de execução. As demais unidades funcionais não são usadas durante a execução do programa *Livermore Loop número 24*.

A execução condicional do programa teste na Configuração 1 (Figura B.19), mantém em atividade as duas ALUs por 28,54% e 6,13% do tempo. A operação nas unidades de acesso à memória de processamento de desvios se dá por 57,22% e 4,08% dos ciclos de máquina. A FPU não é usada durante esse experimento.

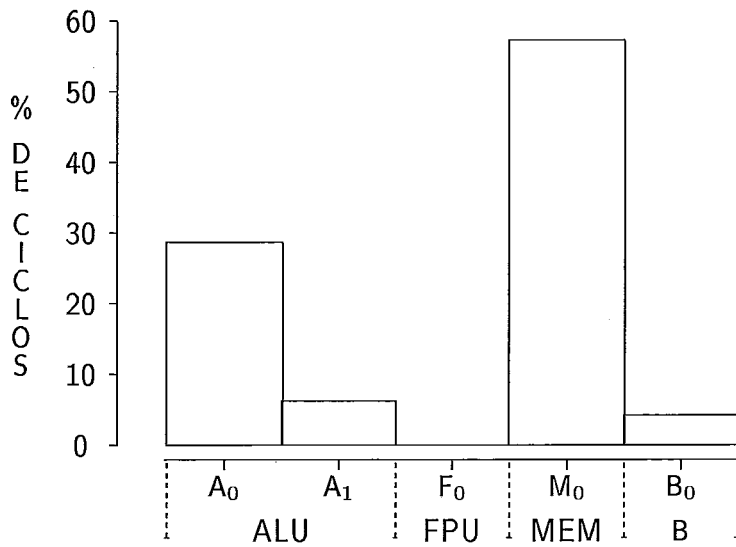


Figura B.19: UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa *Livermore Loop*

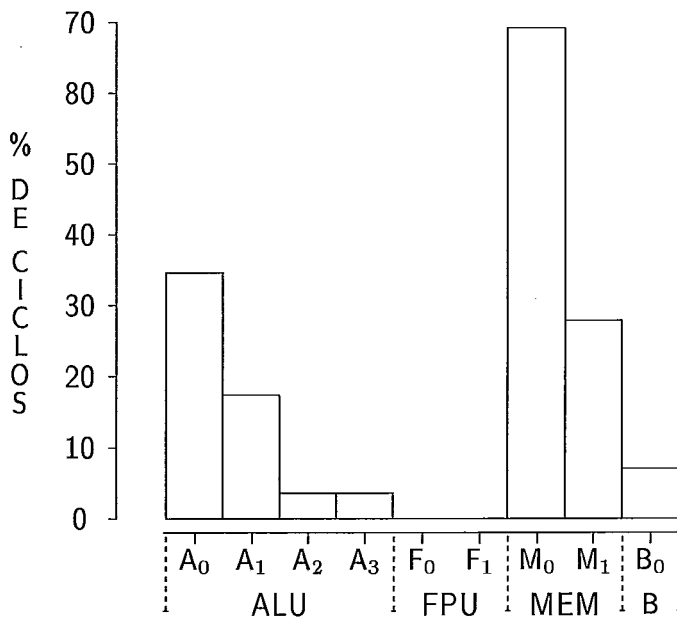


Figura B.20: UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa *Livermore Loop*

Na Figura B.20 vemos o percentual da utilização dos dispositivos funcionais presentes na Configuração 2 do CONDEX, durante a execução condicional do programa *Livermore Loop* número 24. As quatro ALUs processam por 34,43%, 17,21%, 3,45% e 3,45% respectivamente. As duas FPUs não são ativadas e as duas unidades MEMs estão em funcionamento por 69% e 27,68% do tempo de execução. Por

fim, a unidade de processamento de desvios executa 6,90% dos ciclos do processador.

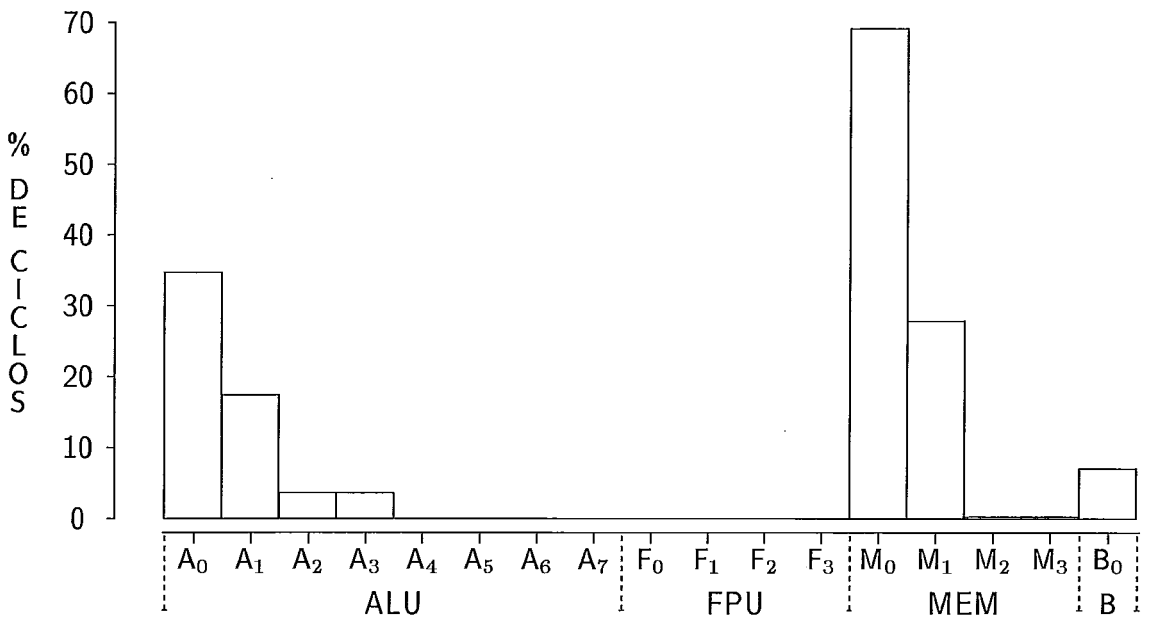


Figura B.21: UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa *Livermore Loop*

Na Configuração 3 a execução condicional, faz uso de quatro das oito ALUs por 34,44%, 17,21%, 3,44% e 3,44% do tempo, como podemos ver no histograma da Figura B.21. As FPUs existentes na configuração não utilizadas e as unidades de acesso à memória executam por 68,95%, 27,62%, 0,04% e 0,04% do tempo necessário para a execução do programa *Livermore Loop* número 24. A unidade de processamento de desvios permanece ocupada por apenas 6,90% dos ciclos do processador.

EXECUÇÃO SEQUENCIAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	363	29,25
2 ALUs	0	0,00
1 FPU	0	0,00
1 MEM	796	64,14
1 BRANCH	82	7,41

Total de IMFs Executadas: 1241

Tabela B.17: UFs Ativas do Programa *Árvore* – Execução Sequencial

Finalmente nas Tabelas B.17, B.18, B.19 e B.20 mostramos a taxa de ocupação de cada uma das unidades funcionais do modelo para cada uma das configurações, quando da execução do programa *Árvore*.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	215	20,97
2 ALUs	74	7,21
1 FPU	0	0,00
1 MEM	796	77,65
1 BRANCH	45	4,39

Total de IMFs Executadas: 1025

Tabela B.18: UFs Ativas do Programa *Árvore* – Configuração 1

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	199	26,18
2 ALUs	52	6,84
3 ALUs	8	1,05
4 ALUs	9	1,18
1 FPU	0	0,00
2 FPUs	0	0,00
1 MEM	352	46,31
2 MEMs	222	29,21
1 BRANCH	45	5,92

Total de IMFs Executadas: 760

Tabela B.19: UFs Ativas do Programa *Árvore* – Configuração 2

Parece-nos desnecessário (e cansativo para o leitor), a repetição completa da análise feita para os quatro programas anteriores. Gostaríamos somente de observar que para o programa *Árvore*, a inclusão de dispositivos funcionais quando evoluímos da Configuração 1 (Tabela B.18), para Configuração 3 (Tabela B.19), é bastante importante para a redução do tempo de execução. Nesse caso os 1025 ciclos de máquina necessários na Configuração 1, ficaram reduzidos a 760 ciclos na

Configuração 2. Menos expressiva foi a aceleração obtida com inclusão das unidades funcionais correspondentes a Configuração 3 (Tabela B.20). Grande parte das novas ALUs permaneceram ociosas durante a execução do programa. Por outro lado as novas MEMs contribuíram significativamente para esse fim, como pode ser concluído com base nos dados contidos na Tabela B.20.

EXECUÇÃO CONDICIONAL		
Nº de UFs Ativas	Nº de IMFs	% do Tempo de Execução
1 ALU	175	26,15
2 ALU	61	9,11
3 ALU	3	0,44
4 ALU	8	1,19
5 ALU	5	0,74
6 ALU	0	0,00
7 ALU	0	0,00
8 ALU	0	0,00
1 FPU	0	0,00
2 FPU	0	0,00
3 FPU	0	0,00
4 FPU	0	0,00
1 MEM	320	47,83
2 MEM	96	14,34
3 MEM	68	10,16
4 MEM	20	2,98
1 BRANCH	45	6,72

Total de IMFs Executadas: 669

Tabela B.20: UFs Ativas do Programa *Árvore* – Configuração 3

Nas Figuras B.22, B.23, B.24 e B.25, temos os histogramas que mostram a atividade das unidades funcionais durante a execução seqüencial do programa *Árvore*, em cada configuração do CONDEX testada. Mais uma vez, apresentamos um único histograma para a execução seqüencial.

Durante a execução seqüencial (Figura B.22), vemos que uma ALU permanece operando por 29,25%, uma unidade de acesso à memória por 64,14% e a unidade de processamento de desvios por 7,14% do total de ciclos de processador necessários para a execução do programa *Árvore*. Os demais dispositivos funcionais não são usados durante os experimentos.

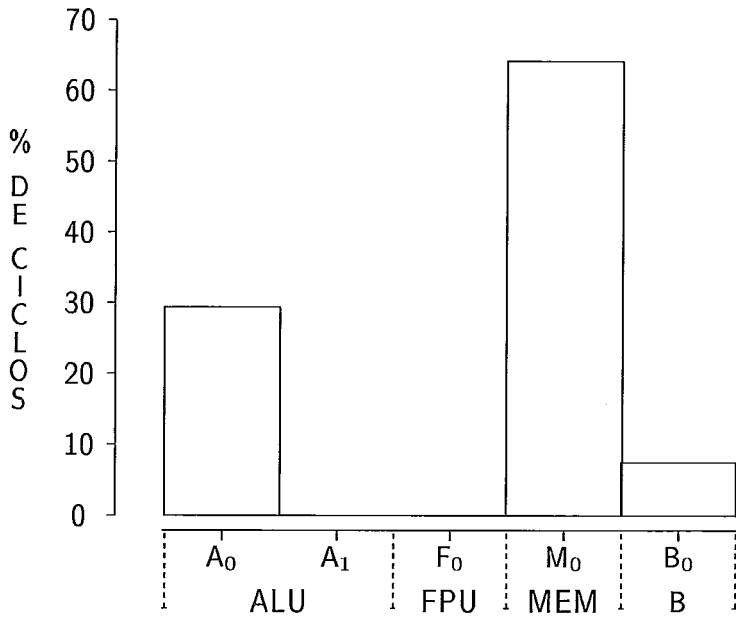


Figura B.22: UFs Ativas na Configuração 1 Durante a Execução Seqüencial do Programa *Árvore*

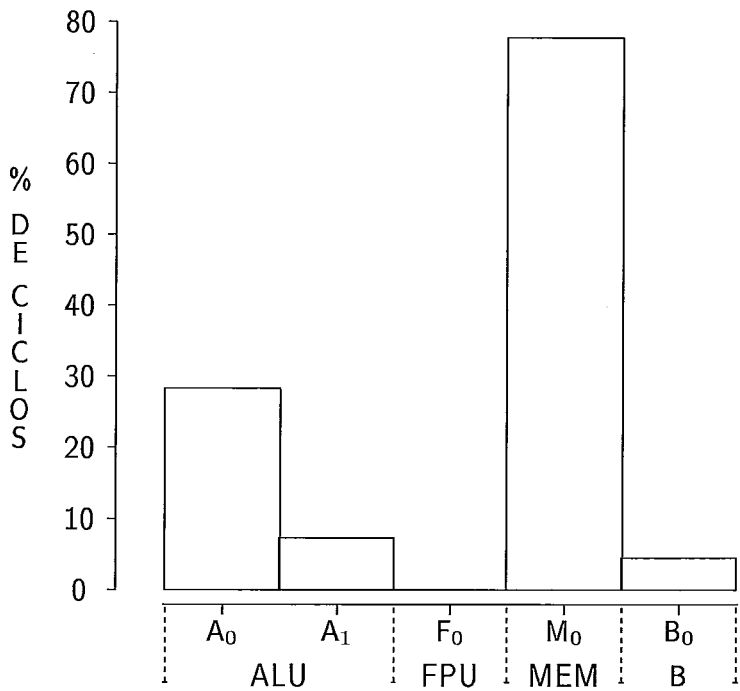


Figura B.23: UFs Ativas na Configuração 1 Durante a Execução Condicional do Programa *Árvore*

Durante a execução condicional na Configuração 1 do nosso modelo de processador, vemos na Figura B.23, que as duas ALUs permanecem ocupadas por 28,18% e

7,21% do tempo total. A única FPU não é usada e as unidades de acesso à memória e de processamento de desvio processam respectivamente por 77,65% e 4,39% dos ciclos de processador.

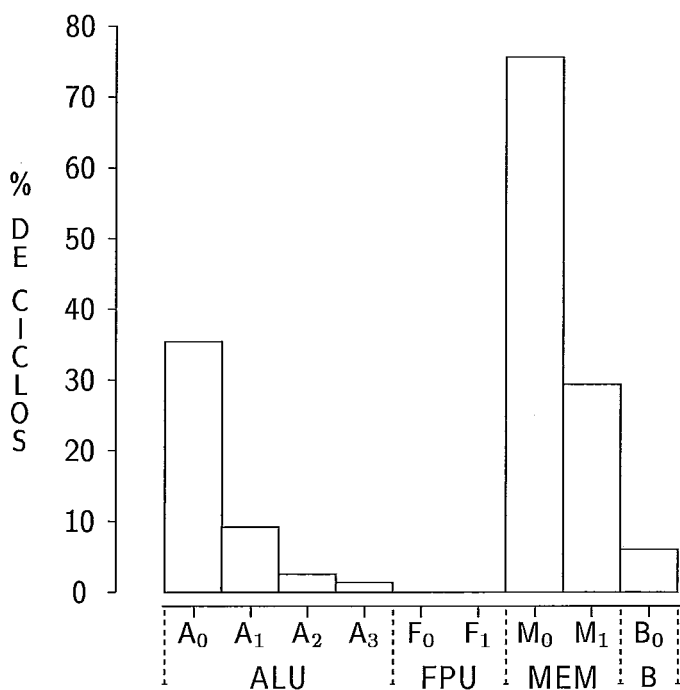


Figura B.24: UFs Ativas na Configuração 2 Durante a Execução Condicional do Programa *Árvore*

Na Figura B.24 temos o histograma que mostra o percentual de ocupação durante a execução condicional do programa *Árvore* na Configuração 2 do CONDEX. As quatro ALUs executam por: 35,25%, 9,07%, 2,23% e 1,18 do ciclos de processador necessários para o teste. As duas FPUs não são acionadas e as duas MEMs operam por 75,52% e 29,21% do tempo. A unidade BRANCH executa por 5,92% do tempo de execução do programa.

A execução condicional do programa *Árvore* na Configuração 3 (Figura B.25), resultou na ocupação de cinco das oito ALUs por respectivamente, 37,63%, 11,48%, 2,37%, 1,93% e 0,74% do tempo. As três ALUs restantes, bem como as quatro FPUs, não são utilizadas. As quatro unidades de acesso à memória e a unidade de processamento de desvios são ativadas por 75,31%, 47,83%, 13,14%, 2,98% e 6,72% do tempo dispendido.

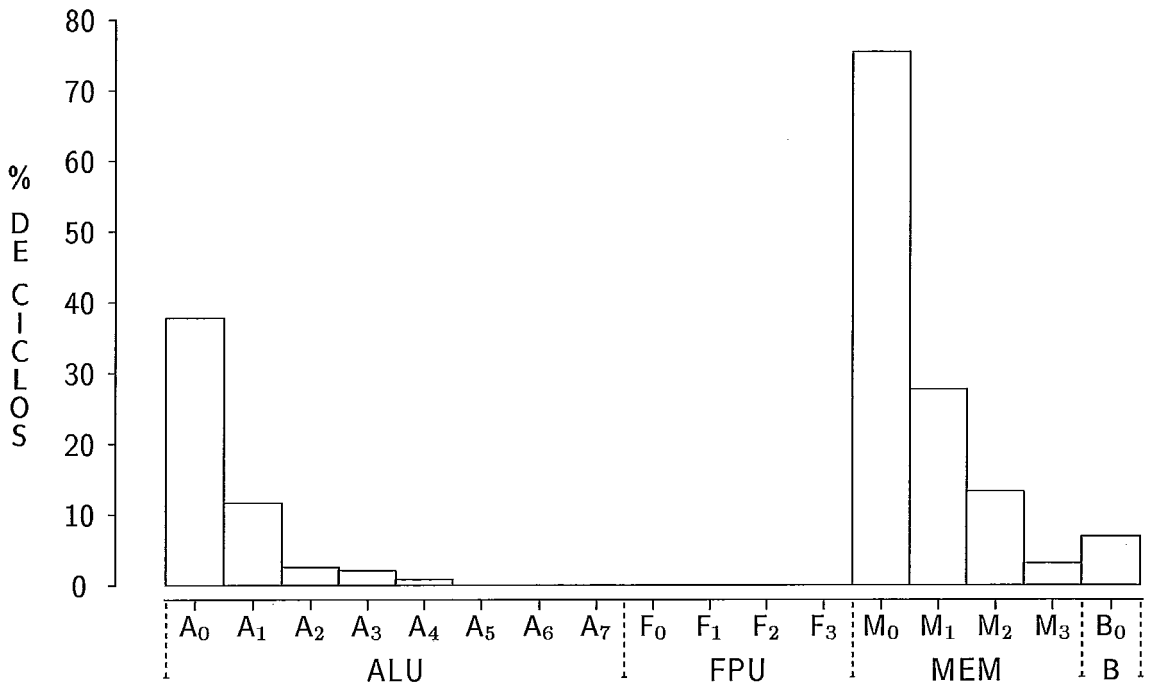


Figura B.25: UFs Ativas na Configuração 3 Durante a Execução Condicional do Programa *Árvore*

A análise de cada um dos programas de teste como foi feita, é certamente útil no desenvolvimento da arquitetura Super Escalar modelada. Embora seja bom frisar, que o dimensionamento do número de unidades funcionais não é o único fator relevante nessa tarefa. O esquema de interconexão dos componentes da máquina, os tempos de latência de cada operação são fatores significativos que devem ser cuidadosamente considerados.

É importante lembrar ainda que a remoção de um dispositivo funcional de um certo tipo poderá provocar um distúrbio no nível de ocupação dos outros componentes do processador. Daí a necessidade do dimensionamento do conjunto como um todo.

Por outro lado as características do programa de aplicação, e a qualidade do código objeto seqüencial correspondente também afetam a qualidade do código compactado e comprometem o desempenho da arquitetura.