Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Marcio de Oliveira Buss

e aprovada pela Banca Examinadora.

Campinas, 22 de Janeiro de 2002

COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Escalonamento de Instruções em Arquiteturas VLIW Particionadas Explorando *Bypassing* de Operandos

Marcio de Oliveira Buss

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Escalonamento de Instruções em Arquiteturas VLIW Particionadas Explorando *Bypassing* de Operandos

Marcio de Oliveira Buss[1]

Agosto de 2001

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Luiz Cláudio Villar dos Santos
  Depto. de Informática e Estatística - UFSC
- Prof. Dr. Ricardo Pannain
  Instituto de Computação - UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação - UNICAMP

[1]**Apoio financeiro CAPES**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Buss, Marcio de Oliveira

B966e Escalonamento de instruções em arquiteturas VLIW particionadas explorando Bypassing de operandos / Marcio de Oliveira Buss -- Campinas, [S.P. :s.n.], 2001.

Orientadores : Guido Costa Souza de Araújo; Paulo C.Centoducatte

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Compiladores (Computadores). 2. Arquitetura de computadores. I. Araújo, Guido Costa Souza de. II. Centoducatte, Paulo Cesar. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Escalonamento de Instruções em Arquiteturas VLIW Particionadas Explorando *Bypassing* de Operandos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcio de Oliveira Buss e aprovada pela Banca Examinadora.

Campinas, 01 de Agosto de 2001.

Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

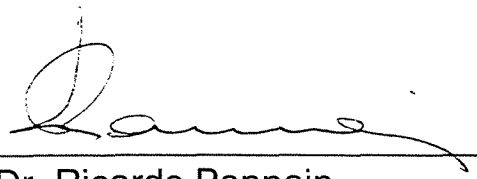
Prof. Dr. Paulo Cesar Centoducatte
(Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

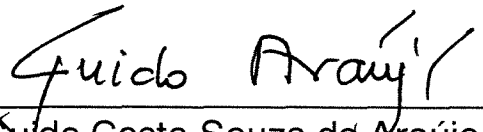
TERMO DE APROVAÇÃO

Tese defendida e aprovada em 01 de agosto de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Luiz Cláudio Villar dos Santos
UFSC



Prof. Dr. Ricardo Pannain
IC - UNICAMP



Prof. Dr. Guido Costa Souza de Araújo
IC - UNICAMP

Resumo

A incansável busca por máquinas mais velozes, aliada aos enormes avanços tecnológicos na concepção de circuitos integrados, retiraram as arquiteturas *Very Long Instruction Word* (VLIW) de um estado amórfico para a realidade. Embora tenham surgido como CIs recentemente [1], as máquinas VLIW foram idealizadas há algumas décadas atrás [13, 16, 22, 23]. Os processadores que definem este modelo de processamento não mais obedecem regras clássicas de execução: instruções de um dos possíveis fluxos de controle de um comando de desvio condicional são executadas mesmo antes do término da avaliação da condição, a qual determinará se a transferência de controle deverá ocorrer ou não; executam simultaneamente inúmeras instruções, de diferentes tipos, oriundas do mesmo programa; computam programas que foram compilados de uma forma revolucionária: todo o programa é analisado em busca de operações paralelizáveis, como se fosse um único (macro) bloco.

Numa tentativa de contribuição a esta linha de pesquisa, este trabalho visa a detecção e exploração do paralelismo 'escondido' em programas originalmente sequenciais. Esta busca gera resultados que são analisados e quantificados com o intuito de se encontrar uma arquitetura-alvo adequada para uma aplicação específica. Esta metodologia encontra-se inserida no contexto de uma área denominada *Embedded Systems*, a qual se preocupa em otimizar ao máximo a execução de uma classe restrita de aplicações ou até mesmo uma única aplicação-chave de um sistema dedicado.

O modelo de arquitetura considerado neste trabalho é denominado VLIW particionado (do inglês *partitioned VLIW*). Este modelo difere da máquina VLIW ideal pelo fato de não possuir um único banco de registradores centralizado, mas sim vários bancos de registradores que se comunicam através de barramentos especiais.

Com este modelo de arquitetura em mãos, o trabalho desenvolvido nesta dissertação trata da investigação de problemas relacionados com o mapeamento de uma aplicação específica a uma máquina VLIW dedicada. Em um macro-cenário, este trabalho tenta responder a seguinte questão: "*Qual é a máquina VLIW adequada para uma dada aplicação?*". Ou ainda, "*Quantos bancos de registradores e quantas unidades funcionais o processador para esta aplicação deveria ter?*".

Abstract

The untiring search for faster machines, allied to the great technological advances in the field of integrated circuits conception, brought out the Very Long Instruction Word architectures from an amorphous status to reality. Although they have appeared recently as real chips [1], the VLIW machines were idealized some decades ago [13, 16, 22, 23]. The microprocessors that define this processing model no longer obey classical rules of execution: instructions coming from one of the possible control flows resulted of a branch instruction are executed even before the finish of the evaluation condition. This evaluation condition will determine if the control transfer should occur or not. Also, these architectures execute simultaneously many instructions, of different kinds, issued from the same program. Moreover, these processors compute programs that were compiled through a revolutionary way: all the program is analized to search for paralelizable operations.

As an attempt to contribute to this research field, this work aim the development of a methodology to detect and exploit the paralelism "hided" in sequential-written programs. The results generated by this search are analized and quantified in order to find a target-architecture for a specific application. This work is inserted in the context of an area called Embedded Systems. This research field worry about the maximum optimization of an application class or even only one key-application of a embedded system.

The architecture model considered in this work is denoted as "Partitioned VLIW Architecture". This model is slightly different of the ideal VLIW architecture model. In the ideal model, there must be only one centralized register file, in order to guarantee the maximum Instruction Level Parallelism (ILP). All the functional units share the same register file. On the other hand, the architecture model being considered here presents many distributed register files, which have an special bus to communicate data among them.

With this architecture model in mind, the work developed in this thesis investigates some of the problems related to mapping one specific application to an embedded VLIW architecture. Roughly speaking, this work tries to answer the following question: "*What is the ideal VLIW architecture for a given application?*" or "*How many register files and how many functional units the processor for that application should have?*".

À todas pessoas que tornaram este trabalho possível.

"Imaginação é mais importante do que o conhecimento."
Albert Einstein.

"Quanto mais você racionaliza, menos você cria."
Raymond Chandler.

Agradecimentos

Agradeço a minha família pelo apoio durante todo este período, bem como durante esses 26 anos que passei até aqui.

Agradeço ao meu orientador, professor Guido Araújo, pela orientação, experiência e amizade valiosas que me foram passadas durante esses dois anos de convivência, desde que cheguei ao Instituto de Computação da UNICAMP. Várias oportunidades surgiram na minha vida acadêmica simplesmente pelo fato de tê-lo como orientador. Obrigado também pelas pizzas dos finais de ano :-)

Ao meu co-orientador, professor Paulo Centoducatte, pelas valiosas discussões e idéias inseridas na dissertação.

Aos professores Ricardo Pannain e Luiz Cláudio dos Santos, por terem aceito o compromisso de revisar este documento, mesmo havendo pouco tempo disponível.

À CAPES, pelo suporte e apoio financeiro concedido durante a realização do mestrado.

A todos os amigos do Instituto de Computação, com os quais pude compartilhar dois valiosos anos de minha vida. Espero poder revê-los qualquer dia desses em lugares completamente inesperados, para que o espanto da surpresa se transforme imediatamente na alegria do reencontro. Especialmente aos colegas do LSC Rodolfo, Sandro e Guilherme por tantas vezes terem auxiliado no desenvolvimento deste trabalho.

Aos amigos de Santa Maria por sempre trazerem um ar mais gaúcho à distante Campinas.

Ao *IMPACT Research Group* da University of Illinois at Urbana-Champaign pela criação do compilador IMPACT e por manter seu código fonte aberto para instituições de ensino e pesquisa como a UNICAMP. Nesta mesma linha de raciocínio, a todas as pessoas que desenvolvem o compilador GnuCC e a biblioteca de edição de executáveis WARTS/EEL, os quais também serviram como plataformas de implementações desta dissertação.

A Deus, pela saúde, força e inesgotável desejo de aprender.

A todos vocês, deixo um grande abraço e o mais sincero muito obrigado.

Marcio.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A classe de arquiteturas conhecida como *Very Long Instruction Word* (VLIW) vem sendo considerada um dos estilos de projetos de processadores mais promissores em termos de aumentar o desempenho computacional para além dos limites das máquinas RISC convencionais. Tais arquiteturas podem tirar vantagem das duas formas possíveis de paralelismo, isto é, temporal e espacial. Em todas as arquiteturas reais paralelismo temporal é tipicamente explorado através da utilização de *pipelining* [31]. Paralelismo espacial em máquinas VLIW é realizado através da execução de mais do que uma operação por ciclo de relógio, utilizando múltiplas unidades funcionais. Isto permite explorar o paralelismo no nível de instrução (ILP)[1] tipicamente "escondido" em programas originalmente seqüenciais.

Em processadores super-escalares, outra classe de arquiteturas que explora paralelismo espacial, a resolução dos conflitos e o escalonamento de instruções são realizados dinamicamente em tempo de execução. O algoritmo mais popular para esta tarefa foi proposto por Tomasulo [63]. Para arquiteturas VLIW, entretanto, o escalonamento e empacotamento das instruções é realizado estaticamente pelo compilador.

Escalonamento em tempo de compilação permite a utilização de análises e transformações sofisticadas que seriam muito caras se realizadas em tempo de execução. Mais ainda, o escalonamento de instruções para máquinas VLIW requer a associação das operações a unidades de execução específicas e imutáveis, assim como a determinação antecipada do ciclo de máquina no qual uma operação será computada. Ou seja, cada operação deve ter suas coordenadas de tempo e espaço definidas antes do início da execução.

A utilização de arquiteturas VLIW no projeto de sistemas é cada vez mais comum. Isto pode ser explicado pela crescente necessidade de desempenho em áreas como telecomunicações e multimídia, bem como pela possibilidade oferecida pelas arquiteturas VLIW de transferirem a complexidade do *hardware* para o compilador. Por exemplo, algoritmos

[1] Do inglês, *Instruction Level Parallelism*

básicos de telecomunicações são tipicamente executados por processadores digitais de sinais (DSPs)[2], e DSPs baseados na filosofia VLIW começam a aparecer no mercado (TI C6201 [34], Philips Trimedia [50], MAP1000 [53] e TigerSharc [25]). Ainda, vários destes DSPs são baseados em uma variação do modelo VLIW ideal denominado *VLIW particionado*. O termo "particionado" refere-se à divisão do banco centralizado de registradores utilizado no modelo "ideal". Este particionamento visa uma redução no número de portas de leitura e escrita no *register file*, reduzindo assim o tamanho do ciclo de relógio [8].

Uma análise do mercado de DSPs pode dar uma idéia de sua evolução. Por exemplo, o lucro total resultante da venda de DSPs para 2003 foi estimado em 13,6 bilhões de dólares [62]. É provável que estes números se tornem ainda maiores com a introdução da telefonia celular de terceira geração (3G) e a difusão da *High Definition Television*, tecnologias altamente "consumidoras" de DSPs.

Por outro lado, o desenvolvimento de *software* para sistemas dedicados é considerado um dos gargalos no projeto de tais sistemas. Como exemplo, a fabricante Siemens anunciou recentemente uma redução de 60% no consumo de potência *stand-by* do telefone móvel C25 simplesmente devido à uma modificação de *software* [61].

Desta forma, essa dissertação tenta dar uma contribuição para a área de pesquisas em técnicas de compilação para arquiteturas VLIW particionadas. Devido às características e fatos apresentados acima, assim como no decorrer desta dissertação, acreditamos que esta organização de arquitetura será o modelo adotado como padrão para as futuras gerações de DSPs.

Neste trabalho abordamos vários tópicos relacionados com o aumento de paralelismo e o escalonamento de instruções para arquiteturas VLIW particionadas. Um algoritmo de escalonamento de instruções baseado em *list scheduling* [44] é proposto. Escalonamento de instruções é a fase mais importante de geração de código para DPSs VLIWs. Esta fase associa unidade funcional e ciclo de relógio específicos para cada operação no intuito de atingir uma máxima utilização do paralelismo disponível.

Ressaltamos aqui que algumas definições foram omitidas desta dissertação por serem consideradas básicas. Nao faz parte do escopo deste trabalho, por exemplo, definir o que são blocos básicos, grafos de fluxo de controle, algoritmos clássicos de escalonamento como *list scheduling*, relações de pós-dominância, execução especulativa, predicação, dependências de dados e seus tipos, traços de um programa, *if-conversion*, etc. Entretanto, o leitor pode encontrar todos estes conceitos nas referências [4] e [44].

[2]Do inglês, *Digital Signal Processors*

1.1 Compiladores para Arquiteturas VLIW

Uma característica crucial para um *compilador* destinado a arquiteturas paralelas (como VLIW) é a de possuir técnicas para detectção e exploração do paralelismo "escondido" em programas originalmente seqüenciais.

Paralelismo é tradicionalmente explorado em duas granularidades: operações individuais são executadas em paralelo nas diferentes unidades funcionais do processador, caracterizando um paralelismo no nível de instrução (*ILP*), e iterações sucessivas de laços são computadas simultaneamente, caracterizando um paralelismo no nível de laços. Compiladores para arquiteturas VLIW devem expor um paralelismo no nível de instrução suficientemente grande para uma utilização efetiva do *hardware*. Estudos prévios têm mostrado que a utilização de técnicas convencionais de otimização e escalonamento de instruções não são suficientes para prover o aumento significativo de desempenho que as arquiteturas VLIW são capazes de proporcionar, principalmente em aplicações com um grande número de instruções de controle [36]. O desenvolvimento de técnicas avançadas de compilação como *Trace Scheduling* [22], *Percolation Scheduling* [45], *Superblock Formation* [33] , *Hyperblock Formation* [43] possibilitam expor o paralelismo no nível de instrução, permitindo com isto um aumento considerável no desempenho dos programas.

Assim sendo, a característica chave de um compilador voltado para uma arquitetura VLIW é a utilização de transformações no programa que tornem possível um aumento significativo do volume de paralelismo.

É consenso que o tamanho médio dos blocos básicos de uma aplicação que apresenta um volume razoável de controle é de cinco instruções [31], geralmente formando uma cadeia de dependências que limita drasticamente a possibilidade de execução paralela de operações. Desta forma, as técnicas que visam explorar ILP devem ir além dos limites de blocos básicos individuais, aumentando assim o espaço de busca do escalonamento. Algumas destas estratégias baseiam-se em informações de *profiling* da aplicação para formar caminhos ou "traços" importantes de execução (do inglês, *traces*) no grafo de fluxo de controle, e a partir daí escalonar as instruções dos traços, os quais atravessam vários blocos básicos. Instruções de compensação [44] devem ser eventualmente inseridas nos blocos fora do traço para que a semântica original do programa seja mantida. Como essas compensações são inseridas em blocos básicos menos executados, o impacto da compensação não é um fator limitante desta técnica. A confiabilidade e precisão do *profiling* tem uma influência razoável no resultado final da compilação. Algoritmos de escalonamento de instruções como *trace scheduling* [22] são fundamentados neste princípio.

Outra técnica que também fundamenta-se nos traços de execução mais freqüentes para o aumento do escopo do escalonamento é a formação dos chamados "super-blocos" [33]. Um super-bloco nada mais é do que um traço sem entradas laterais, ou seja, o caminho

de execução pode entrar somente pelo "topo" do super-bloco (podendo entretanto sair por um ou mais pontos de escape). Além da etapa de formação dos traços, uma técnica conhecida como *tail duplication* [33, 43] é necessária para a composição dos super-blocos.

Todavia, super-blocos não consideram predicação de instruções [51]. Uma estrutura similar ao super-bloco, denominada hiper-bloco, mescla as características do super-bloco com a utilização de predicação. Assim, hiper-blocos combinam instruções provindas de múltiplos caminhos de execução. Desta forma, para programas que não apresentam saltos distantes, hiper-blocos provêem um *framework* mais adequado para a realização das transformações em tempo de compilação.

Além das técnicas baseadas nos traços de execução, o aumento de paralelismo pode ser conseguido através de técnicas de execução simultânea de múltiplas iterações de laços. *Loop-unrolling* [44] é uma técnica que "desenrola" o corpo de um laço pela replicação do seu código, levando em conta as dependências entre as instruções, sejam elas intra ou inter-iterações. O número de vezes que o laço é desenrolado depende do volume de paralelismo desejado e de restrições como tamanho do código final e pressão por registradores. A existência de muitas dependências de dados entre iterações pode aumentar o tamanho do código sem resultar em um aumento significativo de desempenho. Um outro método mais elaborado de sobreposições de iterações de laços é denominado *Software pipelining* [38]. Esta é uma técnica de escalonamento de laços que permite a transferência de instruções de uma iteração para outra enquanto mantém a sobreposição das iterações. Neste caso, porém, o corpo do laço não é desenrolado. Uma das variações de mais aceitas de *software pipelining* é denominada *modulo scheduling* [52]. Esta metodologia é uma abordagem muito eficiente para laços formados por um único bloco-básico, porém não endereça eficientemente aqueles laços que apresentam condicionais em seu interior. Finalmente, métodos como *Perfect Pipelining* [5] combinam *loop unrolling* e a detecção de padrões repetidos para formar o escalonamento final.

Uma outra forma de agrupamento de múltiplos blocos básicos no intuito de aumentar o paralelismo de uma aplicação é a utilização do Grafo de Dependências de Programa (PDG) [21]. O PDG é uma representação de programa que facilita a detecção do paralelismo escondido em programas seqüenciais. Este grafo consegue, de maneira clara e concisa, identificar as instruções do programa que são potencialmente paralelizáveis. Esta representação mostra explicitamente as dependências de controle entre os blocos básicos e as dependências de dados entre operações da aplicação, identificando desta maneira aqueles blocos ditos *controle-equivalentes*. Um aspecto que torna o PDG uma representação poderosa é o fato de que as dependências de dados e controle são representadas no mesmo grafo. Como todas as informações necessárias para avaliar o potencial paralelismo entre duas instruções podem ser facilmente extraídas do PDG, este grafo é particularmente interessante sob o ponto de vista de um compilador que deseja gerar código para uma

arquitetura paralela.

Em termos arquiteturais, pode-se afirmar que existem dois tipos de paralelismos, o paralelismo "ideal" e o "útil". Paralelismo ideal é aquele revelado pelas dependências de controle e pelas dependências de dados de um programa. Paralelismo útil é um sub-conjunto do paralelismo ideal, e representa o que é possível de ser executado simultaneamente em uma dada arquitetura, dadas restrições como número de unidades funcionais e número de bancos de registradores. O volume de paralelismo apresentado pelo "paralelismo ideal" pode não ser compatível com o "paralelismo útil" oferecido pela arquitetura.

1.2 Trabalhos Relacionados

O problema de escalonamento de tarefas/instruções em um conjunto finito de recursos, com conectividade limitada e custos de comunicação diferentes de zero é um tópico bastante estudado e comprovadamente um problema NP-completo [27]. Desta forma, vários algoritmos baseados em heurísticas têm sido propostos com a finalidade de obter uma solução sub-ótima para o problema. Mais especificamente, o problema de escalonamento de instruções realizado em tempo de compilação, tendo-se como arquitetura-alvo uma máquina (VLIW) particionada, tem ganho uma atenção crescente nos últimos anos.

O compilador Bulldog [17] foi a primeira tentativa de geração de código para arquiteturas VLIW particionadas. Associação das operações a *clusters* de unidades funcionais são realizadas hierarquicamente nos traços de execução mais freqüentes da função sendo compilada. *List scheduling* [44] é então aplicado aos traços com a finalidade de ordenamento das instruções. A formação dos traços, a associação das operações aos *clusters* e o escalonamento propriamente dito são realizados em fases distintas.

Mais especificamente, o algoritmo conhecido como *Bottom-Up Greedy Algorithm* (BUG) é utilizado para associar as operações e registradores à *clusters*. Numa primeira fase, BUG atravessa o grafo de precedência de dados desde os nós de saída (folhas) até os nós de entrada (raízes). Nesta travessia, BUG computa uma "estimativa de preferência" dos nós às diferentes unidades funcionais, baseando suas decisões nas localizações dos operandos fontes e destinos já alocados. BUG atravessa o grafo recursivamente e faz uma estimativa sobre a disponibilidade de unidades funcionais e operandos para cada operação. Após todas as operações serem associadas à um traço, o *list scheduler* insere cópias de dados onde necessárias. A maior desvantagem do algoritmo BUG é que ele associa nós a unidades funcionais utilizando-se de "força bruta" (do inglês, *greedy approach*), ou seja, sem conhecer o impacto da associação das operações no comprimento global do escalonamento. Ainda, uma vez que um nó é associado, o *cluster* escolhido torna-se imutável, e a real eficiência da escolha não é avaliada.

Capitanio, Dutt e Nicolau [8, 10, 11] apresentam um algoritmo de associação de operações à *clusters*, ou particionamento, que tem como entrada o código já escalonado para uma máquina VLIW ideal. A arquitetura considerada neste trabalho é denominada *Limited Connectivity VLIW*, ou LC-VLIW, a qual não possui conectividade completa entre os registradores e todas as unidades funcionais. O compilador usado para a geração do código é denominado *Percolation Scheduling Compiler*, desenvolvido na UC-Irvine. Semelhantemente a metodologia adotada em Bulldog, este trabalho utiliza várias fases de compilação. Estas fases são escalonamento, particionamento e recompactação. O código é primeiramente gerado assumindo-se uma máquina completamente conectada (todas as unidades funcionais podem acessar qualquer um dos bancos de registradores). Após esta etapa, o código é particionado para uma arquitetura LC-VLIW e as instruções de cópias necessárias são inseridas. Finalmente, o código é recompactado com a finalidade de diminuir o efeito das operações de cópia no resultado final. O algoritmo usado para o particionamento é similar ao algoritmo LPK [40]. Para o escalonamento das operações o algoritmo utilizado é o RCS (do inglês, *Resource-Constrained Scheduling*) [26]. Uma desvantagem deste método é a característica multi-fase do mesmo: o escalonamento prévio pode resultar em decisões pobres de associação de *clusters*.

Outro trabalho enfocando esta sub-área é encontrado no Multiflow Trace Compiler [13]. Multiflow é baseado nas mesmas idéias implementadas no contexto de Bulldog, apresentando pequenas modificações. Em resumo, os esforços do compilador são voltados para alocar, em um mesmo *cluster*, todas as operações pertencentes a uma mesma cadeia de dependências. Ainda, as cadeias são divididas entre os *clusters* no intuito de aumentar o volume de paralelismo.

O *Algoritmo Unificado de Associação e Escalonamento* (do inglês, *Unified Assign and Schedule Algorithm* (UAS) [48] integra as operações de particionamento e escalonamento em uma única fase. UAS computa a associação de operações a *clusters* no laço mais interno de um *list scheduler*. Essa abordagem evita as desvantagens da metodologia multi-fase adotada em Bulldog e LC-VLIW. O *list scheduler* escalona o máximo de operações possíveis em um ciclo antes de passar para o próximo ciclo. Uma lista de operações-prontas[3], assim como uma lista de *clusters*, são geradas em cada passo. UAS consiste de dois laços principais. O laço mais externo certifica-se de que todas as operações da lista corrente serão escalonadas. O laço mais interno escalona o maior número de operações possível no ciclo corrente. Ainda, o algoritmo mantém um apontador para a operação com a maior prioridade na lista de operações prontas. No caso de haver recursos disponíveis, a operação é escalonada no ciclo atual. Nesta etapa, a associação de *clusters* é executada.

[3]Por operação pronta entende-se uma operação que tenha todas as suas dependências de dados resolvidas.

Primeiramente, é gerada uma lista de prioridade de *clusters* onde a operação pronta pode ser potencialmente escalonada. Nota-se que este algoritmo, portanto, mantém duas listas de prioridades: uma das operações a serem escalonadas e outra dos *clusters* utilizados para o escalonamento. Várias heurísticas são utilizadas na função de prioridades da lista de *clusters*. Cada *cluster* da lista é examinado com a finalidade de determinar se a operação pronta pode ser computada no mesmo. Após um *cluster* disponível ser encontrado, o algoritmo verifica se alguma instrução de cópia é necessária.

Sanchez et. al. [58] valem-se do algoritmo UAS proposto por Özer incrementado de pequenas modificações. Por exemplo, as heurísticas de seleção de *clusters* visam uma minimização do número de comunicações entre *clusters*. Também, com o intuito de aumentar o volume de paralelismo disponível em laços, *modulo scheduling* é aplicado a laços desenrolados (do inglês, *unrolled loops*), ao invés do *list scheduling* encontrado em UAS. Os autores mostram que os laços de certas aplicações-alvo apresentam poucas dependências entre iterações (do inglês, *loop-carried dependences*), e desta forma o algoritmo tende a associar diferentes iterações dos laços aos diferentes *clusters*.

Leupers [42] avalia uma técnica de escalonamento para uma arquitetura VLIW real ao invés de uma arquitetura hipotética. O autor propõe uma técnica composta pos duas fases intercaladas, uma para particionamento e outra para escalonamento. Para o particionamento, a técnica conhecida como recozimento simulado (*simulated annealing*) [37] é utilizada. Para um dado particionamento, um algoritmo convencional de *list scheduling* é aplicado. O custo de escalonamento (número de ciclos) é usado como medida da qualidade do particionamento. Esta informação de retorno é de suma importância para a fase de particionamento, que tenta encontrar uma nova partição para a qual o algoritmo de escalonamento é novamente invocado , e assim por diante. O processo completo é iterado até que um certo critério (temperatura) seja alcançado.

Fisher et. al [18] propõe uma técnica denominada *Partial Component Clustering* para a tarefa de particionamento entre *clusters* de operações representadas por um grafo de fluxo de dados (DFG) [4]. A primeira fase do algoritmo gera os chamados *componentes parciais* adicionando nós desde as folhas dos DFGs seguindo o caminho mais longo até as raízes dos mesmos, até que um certo número máximo de nós seja atingido. Desta forma, o algoritmo adiciona nós aos componentes parciais levando em conta o(s) caminho(s) crítico(s). De fato, uma das idéias principais da técnica proposta é a de associar subgrafos (componentes) do DFG a *clusters* de maneira que instruções de cópia ao longo do caminho crítico sejam evitadas. O critério inicial de associação também leva em conta o balanceamento de instruções (número de nós) por *cluster*. Em uma segunda fase o algoritmo utiliza um simples *list scheduler* para modificar localmente o conjunto inicial

de associações até que o comprimento do escalonamento ou o número de cópias não possa ser mais reduzido. A desvantagem deste método é que o mesmo torna-se eficiente somente para DFGs que apresentam um caminho crítico próximo do menor comprimento do escalonamento (do inglês, *minimum schedule length*). Para DFGs "largos", o caminho crítico é somente um vago *lower bound* do menor comprimento do escalonamento. Neste caso, o número de unidades funcionais se torna o fator limitante.

Özer e Conte [47, 48, 46] propõem um algoritmo provadamente ótimo para o escalonamento de instruções em arquiteturas VLIW particionadas baseado em programação linear inteira. Embora a metodologia não seja prática em um compilador de produção, este trabalho serve como base para estabelecermos um *lower bound* no problema de escalonamento, avaliando o quão próximo do ponto ótimo encontram-se as heurísticas desenvolvidas. Esta formulação pode ser usada em uma estratégia *branch-and-bound* para particionamento e escalonamento simultâneos. Entretanto, o modelo simplificado do processador adotado não torna claro se a técnica é viável para processadores um pouco mais especializados.

Jacome et. al. [35] propõem um algoritmo que serve como ferramenta de exploração prévia requerida no projeto de *datapaths* especializados de arquiteturas particionadas. Ao invés de realizar uma exploração pela experimentação de várias configurações de arquitetura pré-estabelecidas, os autores propõem uma metodologia que tenta encontrar, para cada aplicação, o número de *clusters* adequado da arquitetura. A arquitetura encontrada para um dada aplicação pode ser a mais inesperada possível, como por exemplo cinco *clusters* contendo, cada um, unidades funcionais de diferentes tipos e quantidades. Esse tipo de metodologia é denominada *clusterização* não-homogênea[4], ou seja, o número e tipos das unidades funcionais em cada *cluster* não é necessariamente o mesmo. O algoritmo proposto pelos autores é baseado nas chamadas agregações horizontais e verticais, que são aplicadas no intuito de encontrar *clusters* em sub-DFGs. Sub-DFGs constituem "pedaços" ideais de computação que podem ser realizadas em um único *datapath*, não requerendo inter-comunicações com outros componentes.

Abnous [2, 3] baseia-se em RCS (*Resource Constrained Scheduling*) para realizar o escalonamento de instruções em uma arquitetura VLIW não-particionada, ou seja, que possui um único banco de registradores. Embora esse trabalho não seja totalmente relacionado com esta dissertação, o estudo detalhado do *bypassing* em arquiteturas VLIW em [2, 3] torna sua contribuição especialmente útil. Mais especificamente, nesta dissertação identificamos que o problema de reduzir *stalls* no *pipeline* pela reassociação de operações a diferentes unidades funcionais (abordado por Abnous) é extremamente relacionado com

[4]Do inglês, *non-homogeneous clustering*.

o problema de redução do número de cópias entre bancos focalizado aqui. Após escalonar as operações utilizando-se de *Resource Constrained Scheduling*, Abnous realiza uma modificação na alocação das instruções longas aos *clusters* tentando todas as permutações possíveis das operações contidas na instrução longa. Aquela que resultar em um menor número de *stalls* no *pipeline* é escolhida. No capítulo 2 abordamos detalhadamente esta questão.

Topham [20] propõe uma arquitetura VLIW particionada diferente da abordada nesta dissertação. Em [20] os bancos de registradores são organizados sob a forma de uma fila, onde somente os bancos vizinhos possuem comunicação via instrução de cópia de operandos. Caso um dado seja requerido em um banco "B", distante do banco de origem "A", cópias intermediárias devem ser inseridas a fim de fazer o dado "correr" na fila. Embora a arquitetura VLIW seja particionada, este trabalho é o único a sugerir este tipo de organização.

1.3 Organização da Dissertação

Essa dissertação está dividida em seis capítulos, organizados da seguinte forma:

- **Capítulo 2:** Este capítulo apresenta as características das arquiteturas VLIW e VLIW particionadas.
- **Capítulo 3:** Os métodos utilizados para aumentar o volume de paralelismo citados na introdução desta dissertação são aprofundados neste capítulo. Abordamos também a utilização do Grafo de Dependências do Programa (PDG) como ferramenta para explorar o paralelismo "escondido" em aplicações escritas seqüencialmente.
- **Capítulo 4:** Este capítulo traz uma análise das definições das variáveis (registradores) em programas retirados de vários *benchmarks*, justificando a eficiência da utilização de *bypassing* de operandos em arquiteturas VLIW.
- **Capítulo 5:** Nesse capítulo apresentamos uma proposta para realizar o escalonamento de instruções em arquiteturas VLIW particionadas enquanto consideramos a possibilidade de *bypassing* de operandos entre unidades de execução pertencentes a diferentes *datapaths*. Mostramos também um conjunto de experimentos que dão suporte a esta abordagem.
- **Capítulo 6:** Nesse capítulo final, apresentamos os principais problemas encontrados na implementação dos experimentos, apontando algumas possíveis soluções para os mesmos. Encerramos com uma conclusão final e descrição de possíveis trabalhos futuros.

Capítulo 2

Arquiteturas VLIW-Particionadas

2.1 Arquitetura VLIW

Arquiteturas VLIW são arquiteturas que executam instruções denominadas "instruções longas", cada uma formada por várias operações independentes. Essas operações são carregadas simultaneamente e distribuídas entre as várias unidades funcionais disponíveis na arquitetura. O escalonamento das operações é pré-determinado pelo compilador, que tem a responsabilidade de atribuir, para cada operação, a unidade funcional e os ciclos de relógio nos quais a operação será executada. Uma máquina VLIW típica possui uma única unidade de controle que lê uma instrução (longa) por ciclo de relógio. Cada operação consume um número pré-estabelecido de ciclos.

As instruções longas podem ser computadas utilizando-se a técnica conhecida como *pipeline*. Neste caso, há um *datapath* executando cada operação que compõe a instrução longa. Máquinas VLIW apresentam um único fluxo de execução, isto é, um único contador de programa (PC), assim como um grande número de *datapaths* e unidades funcionais cuja utilização é totalmente planejada em tempo de compilação. O custo de cada operação é exposto no nível do conjunto de instruções, de forma que o compilador pode otimizar o escalonamento de instruções de maneira eficiente. Outra característica desejada, porém não obrigatória, é a execução predicada [49]. Neste modelo de processamento, instruções de controle são convertidas em dependências de dados, que se manifestam através de predicados associados às instruções. Estas, por sua vez, têm seus resultados efetivados somente se o predicado associado às mesmas for igual ao predicado do caminho do fluxo de controle tomado durante a execução.

2.1.1 Implementação do Modelo VLIW

Microprocessadores baseados no modelo VLIW podem ser implementados de várias formas diferentes, desde o projeto completo de unidades funcionais heterogêneas até a composição de uma arquitetura VLIW através da replicação de *pipelines* RISC padrão. A forma final da arquitetura VLIW será ditada pela relação "custo x desempenho" quando consideradas as possíveis alternativas de implementação e o nicho de aplicação sendo analisado. Entretanto, o objetivo máximo do modelo VLIW é o de eliminar o complicado despacho e escalonamento de instruções que ocorre dinamicamente (portanto via *hardware*) em processadores super-escalares.

Desta forma, os processadores VLIW possuem uma lógica de controle relativamente simples devido ao fato de não realizarem o escalonamento nem o reordenamento dinâmico de instruções. Estas tarefas são realizadas estaticamente pelo compilador.

Cada instrução (longa) do conjunto de instruções das máquinas VLIW tende a ser formada por composições de operações simples (ou instruções "*RISC-like*"). O compilador deve compactar várias operações primitivas em uma única instrução longa, de forma a maximizar o número de unidades funcionais em uso. Isso requer um volume significativo de paralelismo no nível de instrução (ILP) do programa, a fim de preencher eficientemente os *slots* de execução disponíveis.

Uma das desvantagens do modelo VLIW é a necessidade de um compilador extremamente poderoso que possibilite uma eficiente utilização dos recursos de *hardware* disponíveis. Além disso, devido às técnicas agressivas de escalonamento imprescindíveis neste modelo, o tamanho do código-objeto gerado para os programas tende a aumentar significativamente.

Para prover operandos para as várias unidades funcionais paralelas, exige-se ainda uma banda passante do(s) bancos(s) de registrador(es) que deve ser suficientemente grande para não limitar o desempenho final do processador.

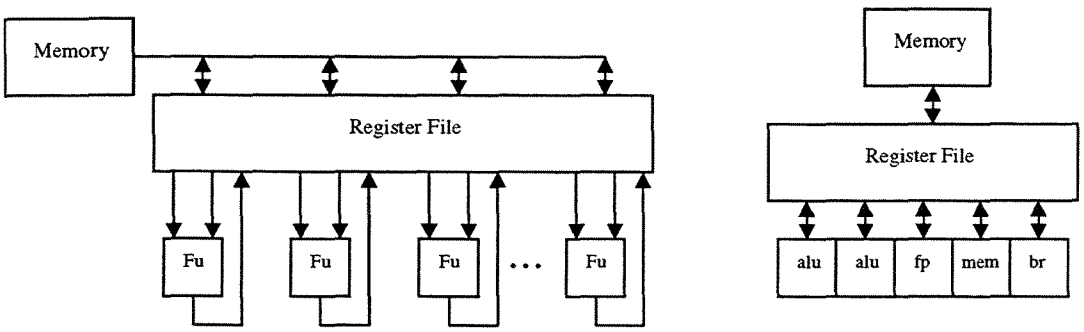
Compatibilidade entre gerações de processadores VLIW é outra desvantagem deste modelo: devido ao fato do compilador ter que "conhecer" profundamente o *hardware* a fim de extrair o máximo desempenho do mesmo, a dependência entre o código gerado e os detalhes do *hardware* é muito forte.

Outra característica-chave relacionada às máquinas VLIW diz respeito ao modelo de banco de registradores empregado. Como será exposto mais adiante, a máquina VLIW ideal possui algumas peculiaridades que a torna impraticável com a atual tecnologia. Uma destas limitações é referente ao banco único de registradores requerido por esta máquina "ideal". Cada unidade funcional tem, idealmente, duas portas de leitura e uma porta de escrita. Assim sendo, o banco de registradores (centralizado) deve prover uma banda passante suficiente para suprir dois operandos por unidade funcional e realizar uma operação de escrita por unidade por ciclo de relógio. Com um número elevado de

unidades funcionais, o número total de portas requeridas pelo banco de registradores torna-o demasiadamente grande e lento. Isso aumenta o custo (maior área) e o tempo de ciclo do processador.

Outra característica que diferencia as máquinas VLIW entre si diz respeito ao número e tipos de unidades funcionais encontradas na arquitetura. Um processador VLIW pode ser formado pela replicação de vários *datapaths* idênticos, caracterizando uma arquitetura homogênea, ou ser constituído por várias unidades funcionais diferentes, que executam diferentes tipos de operações, formando uma arquitetura heterogênea. Neste caso, um estudo prévio baseado em *benchmarks* compostos por programas representativos das aplicações-alvo deve ser realizado com o objetivo de determinar quais os tipos, e em que quantidades, as unidades funcionais serão implementadas. Abnous [2, 3] apresenta um estudo de uma arquitetura denominada VIPER, que possui quatro unidades de aritmética e lógica, duas unidades de comunicação com a memória e duas unidades de transferência de controle.

As diferentes formas de organização de processadores VLIW são comumente representadas conforme mostrado na figura 2.1.



a. Replicação de vários datapahts idênticos.

b. Unidades funcionais distintas.

Figura 2.1: Diferentes organizações de arquiteturas VLIW.

2.2 A máquina VLIW ideal

Com a finalidade de atingir um alto grau de paralelismo no nível de instrução, a máquina VLIW ideal possui várias unidades funcionais homogêneas conectadas a um único banco centralizado de registradores de propósito-geral. Esta topologia possibilita que qualquer operação seja executada em qualquer uma das unidades funcionais disponíveis, assim como ter acesso à qualquer registrador do banco. O modelo VLIW ideal pode ser visto como um dispositivo horizontal de microcódigo que pode decodificar e executar diferentes

opcodes em diferentes unidades funcionais. Cada unidade funcional é completamente equivalente a todas as outras, de forma que não existam dependências estruturais entre elas. Neste modelo de arquitetura, o banco centralizado de registradores deve prover duas portas de leitura e uma porta de escrita para cada uma das N unidades funcionais. Conseqüentemente, a banda passante requerida é bastante considerável ($2N$ leituras e N escritas simultâneas).

Outra informação importante, que nem sempre é considerada na literatura a respeito da arquitetura VLIW ideal, é o nível de inter-comunicação direta entre diferentes *datapaths* via *bypassing* (ou *forwarding*) de operandos [42, 47, 20]. *Bypassing* é um mecanismo usualmente adotado para passagem de operandos entre diferentes estágios do *pipeline* de um único *datapath*. Todavia, este mecanismo pode também ser empregado num âmbito maior, ou seja, inter-*datapaths* de uma arquitetura paralela.

A rede de *bypassing*, se empregada em um âmbito inter-*datapaths*, permite a transparência de dados (operandos) de um estágio do *pipeline* de um dado *datapath*, onde ele é conhecido, para um outro estágio do pipeline de um outro *datapath*, onde ele é requerido. Entretanto, segundo [19], *bypassing* entre unidades funcionais distintas não é adotado como padrão no conceito de arquiteturas VLIW.

Esta dissertação introduz na definição da máquina VLIW ideal a medida do "volume" de inter-comunicação direta entre os diferentes *datapaths* da arquitetura. Um processador VLIW ideal deve possuir uma rede de *bypassing* de operandos que permita uma transferência de dados de qualquer estágio do *pipeline* (entre os estágios de execução e *write-back*) de um determinado *datapath* para o estágio de execução de qualquer outro *datapath*. Embora esta característica não seja citada na definição de arquiteturas VLIW ideais encontradas na literatura, veremos mais adiante que a falta de *bypassing* de operandos entre diferentes *datapaths* pode levar a uma degradação considerável no desempenho das arquiteturas VLIW.

A figura 2.2 mostra um diagrama esquemático da máquina VLIW ideal sendo considerada nesta dissertação.

Uma característica que pode ser observada no modelo VLIW ideal é a simplificação significativa na fase de geração de código realizada pelo compilador. O escalonamento de instruções para uma máquina com um banco de registradores centralizado, unidades funcionais homogêneas e total intercomunicação (via *bypassing* de operandos) entre unidades funcionais não precisa levar em conta qual unidade funcional será atribuída à operação sendo escalonada. Basta verificar se existe alguma unidade disponível e se a operação tem todas as dependências resolvidas.

Entretanto, à medida que o número de unidades funcionais da arquitetura cresce, cresce também o número de portas de leitura e escrita necessárias ao banco de registradores central e a rede de *bypassing* de operandos, aumentando o custo e diminuindo o desempe-

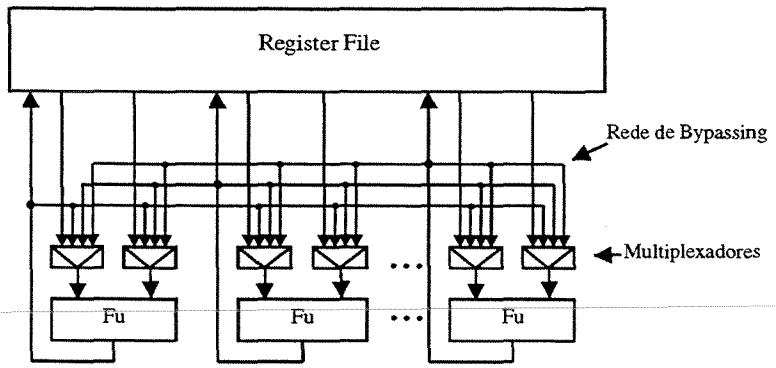


Figura 2.2: Máquina VLIW ideal.

nho da arquitetura. Uma solução bastante aceita para este problema é tratada na seção seguinte.

2.3 A Máquina VLIW Particionada

O modelo de arquitetura adotado como base para os estudos desta dissertação é denominado arquitetura VLIW particionada[1]. Uma arquitetura VLIW particionada contém vários agrupamentos, estruturalmente idênticos ou algumas vezes levemente diferentes, cada um formado por um banco local de registradores e uma ou mais unidades funcionais. As unidades funcionais pertencentes a um mesmo *cluster* têm acesso eficiente ao banco de registradores associado ao *cluster*. Para o acesso a registradores que não estejam alocados no banco de registradores do *cluster*, uma cópia explícita a partir do *cluster* origem deve ser realizada durante a execução do programa.

As arquiteturas particionadas, se homogêneas, são caracterizadas pelo número total de unidades funcionais, F, e o número total de bancos de registradores, R. O número de unidades funcionais por *cluster* é computado como sendo F / R.

Num contexto de sistemas dedicados e processadores especializados, o número total de unidades funcionais e o número total de banco de registradores deve ser ajustado a fim de equalizar o paralelismo disponível em uma aplicação com o paralelismo oferecido pela arquitetura. O termo *largura* é comumente utilizado para denotar o número de unidades funcionais que compõem uma determinada configuração. Geralmente, as possíveis larguras são limitadas a potências de 2, porém pode-se realizar uma exploração arquitetural que não leve em conta esta restrição. Neste caso, certas aplicações podem resultar em configurações atípicas, como por exemplo três *clusters* com um número heterogêneo de

[1]Do inglês, *clustered VLIW architecture*

unidades funcionais por *cluster*. Nesta dissertação, todavia, as arquiteturas consideradas apresentam larguras limitadas a potências de 2 e todas as partições apresentam o mesmo número de unidades funcionais que podem executar qualquer tipo de operação. A razão para isso provém do fato de que uma máquina com muitas restrições e idiossincrasias geralmente aumentam a dificuldade na geração de código realizada pelo compilador. Entretanto, a generalização da arquitetura para unidades funcionais que executam somente um número limitado de tipos de operações, ou até mesmo somente um tipo de operação, é uma alteração simples de ser realizada.

A figura 2.3 mostra a organização de uma arquitetura VLIW particionada.

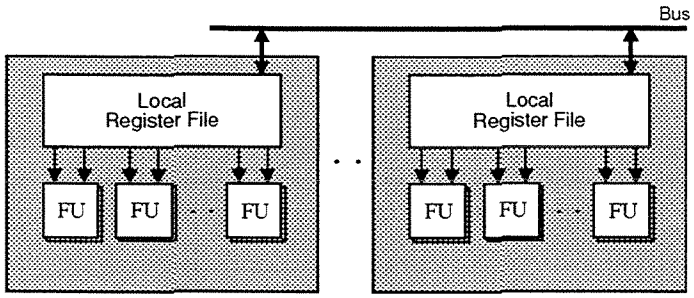


Figura 2.3: Arquitetura VLIW particionada.

Capitanio et. al. [9] apresentam uma análise detalhada do comportamento da área e do tempo de acesso de um banco de registradores quando varia-se o número de portas do mesmo. Os autores mostram que, para um número razoável de portas, o tempo de acesso é proporcional ao logaritmo do número de portas de saída. A complexidade da área é representada por uma função que cresce com o quadrado do número total de portas, isto é, portas de entrada mais portas de saída. Rixner [54] mostra ainda uma relação direta entre *área* $\times$ *tempo de acesso* $\times$ *dissipação de potência* e o número de *unidades funcionais* conectadas a um banco de registradores. Para N unidades funcionais conectadas, a área do banco cresce numa taxa proporcional a N^3, o atraso a $N^{\frac{3}{2}}$ e a dissipação de potência a N^3.

Desta forma, uma implementação prática de uma máquina VLIW com várias unidades funcionais deve executar código paralelo particionado em múltiplos *clusters*, cada um apresentando um limite no número de portas no banco de registradores local.

O particionamento do código em uma arquitetura com múltiplos bancos de registradores pode resultar numa degradação do desempenho devido aos movimentos de dados extras necessários para transferir operandos de uma partição para outra. Se as decisões de escalonamento forem tais que duas operações dependentes A e B são alocadas a unidades funcionais de diferentes *clusters*, transferências de dados extras devem ser previstas

e despachadas pelo compilador a fim de que a(s) dependência(s) de dado(s) entre A e B sejam resolvidas. Em contrapartida, com bancos de registradores menores e mais rápidos, o ciclo de relógio pode ser reduzido aumentando a freqüência de operação do processador. A solução de compromisso entre o número de unidades funcionais conectadas por banco de registradores e o tamanho do ciclo de relógio deve ser analisada. Isto é, o número de instruções de cópia em uma configuração que apresenta vários bancos de registradores tende a ser maior, resultando em um maior número de ciclos para completar determinada aplicação. Por outro lado, o tamanho do ciclo de relógio nesta configuração certamente será menor, tendendo a diminuir o tempo total de execução. Como o desempenho total deve ser medido em termos do tempo de execução, nem sempre a configuração que resulta em um menor número de ciclos será a mais rápida. Dutt et. al. [8] traz um estudo de diferentes configurações para uma arquitetura VLIW particionada e o respectivo impacto no tamanho do ciclo de relógio.

Um exemplo de arquitetura VLIW particionada é mostrado na figura 2.4. Neste exemplo, denominado *Limited Connectivity VLIW* [8], barramentos especiais são destinados a inter-comunicação entre as diferentes partições. O número total de portas por banco de registradores é 12, sendo 4 portas de saída, 2 portas de entrada e 4 portas adicionais de entrada destinadas a comunicação entre *clusters*.

Numa instrução de cópia, o dado é lido por uma das porta de saída do banco *origem* e conduzido até uma das portas especiais de entrada do banco *destino*. Neste intervalo de tempo, a unidade funcional que está conectada à porta de saída do banco origem é posta inativa. O mesmo não se pode fazer com a unidade funcional do banco destino da cópia devido ao fato de que esta unidade funcional pode estar executando uma operação válida.

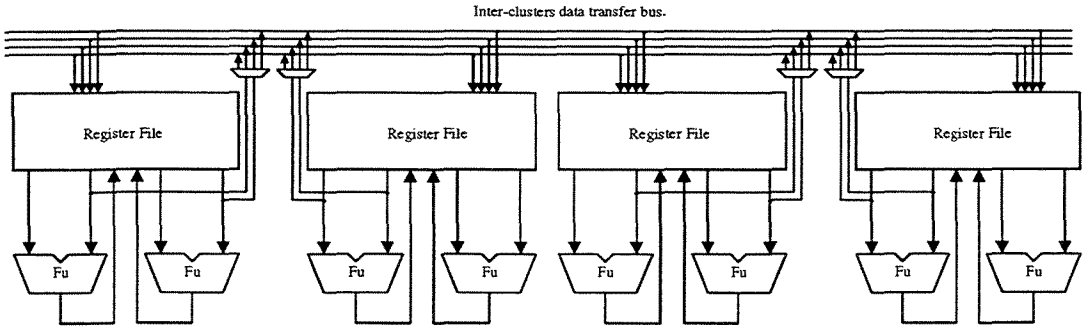


Figura 2.4: Modelo de arquitetura VLIW constituída de diferentes *clusters* proposta por Dutt et. al.

Assim, uma das tarefas adicionais do compilador é a de inserir instruções de cópia para resolver dependências de dados entre operações escalonadas em unidades funcionais pertencentes a diferentes partições. Esquemas em *hardware* também podem ser considerados para esta tarefa. Algoritmos e circuitos de coerência de *cache* podem ser adaptados para

esta arquitetura no intuito de minimizar a complexidade do compilador. Em qualquer dos casos, geralmente considera-se que a cópia de uma partição para outra leva um ciclo para ser completada. Além desta, não se considera qualquer latência extra para estas operações quando está computando-se o escalonamento de operações cujos operandos estão armazenados em diferentes *clusters*.

O modelo de arquitetura considerado nesta dissertação é basicamente o apresentado na figura 2.4. Uma pequena diferença é apresentada na seção 2.3.2.

Entretanto, os dois modelos são *load-store*, ou seja, todas as computações são realizadas em registradores de propósito geral, não havendo instruções com operandos provenientes diretamente da memória. Desta forma, a arquitetura não impõe uma alocação de registradores implícita durante a fase de seleção de instruções [7].

Pipeline é utilizado no *datapath* de todas as unidades funcionais pertencentes a cada *cluster*. Assim, pode-se explorar paralelismo temporal com o propósito de aumentar o desempenho da aplicação. Entretanto, devido à falta de *bypassing* completo (possivelmente) encontrado nas arquiteturas VLIW, o *pipeline* utilizado pode sofrer *data hazards*, mesmo em processadores não-particionados. Este problema é tratado na seção a seguir.

2.3.1 *Pipeline* e *Bypassing* em Arquiteturas VLIW

Vários trabalhos endereçam o mecanismo de *bypassing* no contexto de processadores VLIW [2, 14, 65, 56, 55]. Em [2], os autores analisam as medidas de desempenho de vários esquemas de *bypassing* em termos de sua eficiência na resolução de *data hazards* em uma máquina VLIW com *pipelines* de quatro e cinco estágios. Yung [65] apresenta uma pequena *cache* de registradores em conjunto com a técnica de *bypassing* no intuito de aumentar o desempenho de uma arquitetura chamada *Register Scoreboard and Cache*. Nesta metodologia, os resultados a serem fornecidos pela rede de *forwarding* podem também ser procedentes da *cache* proposta.

Com a finalidade de resolver *data hazards* provenientes da adoção de *pipeline*, processadores RISC utilizam uma técnica conhecida como *bypassing* ou *forwarding*. A função do *bypassing* é resolver *data hazards* que vêm à tona quando uma instrução "B" necessita do resultado proveniente de uma instrução "A" no *pipeline* que ainda não foi escrito no banco de registradores no momento em que a instrução "B" lê seus operandos-fonte. A técnica de *forwarding* utiliza-se dos registradores do *pipeline* para passar os resultados de instruções já executadas diretamente de volta às unidades funcionais em que esses valores são requeridos. Em processadores RISC, o custo do *bypassing* é ínfimo se comparado ao ganho com relação ao número de ciclos. Se definirmos d como sendo o número de estágios de *pipeline* entre a fase de decodificação e a fase de *write-back*, $2d$ comparadores são necessários para a resolução dos *hazards*. Além disso, o número e comprimento das linhas de

roteamento necessárias para a comunicação entre os registradores do *pipeline* e as entradas da unidade de execução são restritas a um *datapath*, geralmente conduzindo a uma solução praticável. Se o circuito de *bypassing* for cautelosamente projetado, o *overhead* resultante no ciclo de clock é minimizado, e a utilização desta técnica é altamente vantajosa.

Entretanto, em processadores VLIW com várias unidades funcionais, a técnica de *bypassing* pode se tornar tão custosa a ponto de se tornar impraticável. Uma máquina VLIW com n unidades funcionais requer um número de comparadores igual a $2dn^2$. Felizmente, os comparadores citados não estão no caminho crítico que limita o comprimento do ciclo de *clock*. Isso porque as comparações necessárias para a verificação do *bypassing* podem começar logo após o início do estágio de decodificação das instruções, e seguir em paralelo com a própria execução da instrução. Todavia, o grande número de comparadores necessários em uma máquina com muitas unidades funcionais e vários estágios de pipeline por unidade funcional pode apresentar um custo significativo na área de silício requerida. Essa porém não é a maior restrição. Com as atuais tecnologias de 0.18 micra, "pênaltis" em área desta magnitude podem ser contornáveis. A maior e mais contundente restrição à técnica de *forwarding* nestas arquiteturas é o número de barramentos necessários para realizar o fluxo dos operandos entre as diversas (e eventualmente distantes) unidades funcionais. Esses barramentos não somente representam problemas globais de roteamento como também podem pertencer ao caminho crítico do circuito integrado, pois são linhas globais conectando múltiplos *datapaths* e certamente apresentam uma alta capacitância.

A tabela 2.1 mostra números relativos a quantidade de barramentos de passagem de operandos e comparadores necessários quando o número de unidades funcionais de um processador VLIW variam. Note que este problema não é restrito às arquiteturas VLIW particionadas, mas sim uma característica intrínseca de um modelo de arquitetura composto por várias unidades funcionais.

n	d	Barramentos	Comparadores
4	2	32	64
8	2	128	256
16	2	512	1024
32	2	2048	4096

Tabela 2.1: Número de barramentos e número de comparadores necessários para um *bypassing* total em arquiteturas VLIW quando varia o número de unidades funcionais.

Na tabela 2.1, *d=2* refere-se a um *pipeline* de 5 estágios, ou seja, dois estágios entre o estágio de decodificação e o estágio de *write-back*.

Desta forma, o *bypassing* a ser implementado em uma arquitetura VLIW deve ser tal

que o *overhead* de área e aumento do comprimento do ciclo de relógio não comprometam o desempenho global do processador. Abnous [3] mostra um estudo amplo e detalhado a respeito da técnica de *bypassing* aplicada a um processador denominado VIPER (VLIW Integer Processor), idealizado na UC Irvine. Neste trabalho, várias alternativas de *bypassing* são experimentadas no intuito de se verificar a quantidade ideal de inter-comunicação entre as diferentes unidades funcionais de VIPER. Os autores concluem que o *bypassing* deve ser restrito a, no máximo, unidades funcionais vizinhas e que uma comunicação completa entre todas as unidades compromete o desempenho e a área do processador.

Como VIPER não provê *bypassing* completo, isto é, de qualquer unidade para qualquer unidade, *data hazards* podem acontecer. *Data hazards* do tipo *Read-After-Write* que não puderem ser resolvidos pela rede de *forwarding* disponível devem ser detectados em tempo de execução pelos próprios comparadores de *bypassing*, e resolvidos pela inserção de uma bolha (*stall*) no *pipeline*. Embora VIPER não possua *barramentos* suficientes para transferir operandos entre quaisquer unidades funcionais, a quantidade de *comparadores* é tal que todos os *hazards* que vierem a ocorrer possam ser detectados. Entretanto, nem todos poderão ser resolvidos pela rede de *bypassing* de operandos.

Um resultado importante extraído desta abordagem diz respeito ao reordenamento das operações nas unidades funcionais com o objetivo de reduzir o número de *stalls* no *pipeline*. Se duas operações geram um *data hazard* devido à escolha das unidades funcionais para sua execução, uma redistribuição horizontal das mesmas, alocando-as à unidades comunicantes, pode impedir a "bolha" no *pipeline*. Esta característica é tratada através da inserção de um módulo de reordenação de instruções no compilador adotado para VIPER.

Note, portanto, que o problema de associar operações a unidades funcionais com a finalidade de reduzir o número de *stalls* no *pipeline* é extremamente similar ao problema de reduzir o número de cópias necessárias em arquiteturas particionadas. Mais ainda, o problema de redução de cópias em arquiteturas particionadas pode ser transformado em um problema de redução do comprimento total do escalonamento, como abordado no capítulo 5.

2.3.2 Análise do Fluxo das Instruções no *datapath* de uma Máquina VLIW Particionada e Não-Particionada

Em um processador VLIW não-particionado (com um único banco de registradores), a tentativa de reduzir o número de *data hazards* é realizada através do escalonamento de operações dependentes em unidades funcionais comunicantes. Quando duas operações dependentes são escalonadas em ciclos sucessivos, em unidades distintas e não-comunicantes, *stalls* no *pipeline* deverão ser induzidos (em tempo de compilação, via instruções de *NOP*). Suponha, por exemplo, que duas operações dependentes A e B foram escalonadas em ci-

clos sucessivos e atribuídas a diferentes unidades funcionais que utilizam somente o banco de registradores central para comunicação. Estas duas operações podem ser vistas na figura 2.5. *R2* e *R7* são os registradores destino nas operações mostradas.

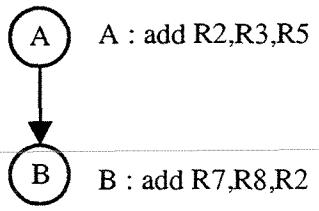


Figura 2.5: Operações dependentes A e B.

Como a arquitetura sendo considerada possui *pipeline* em cada *datapath*, o resultado da operação A, usado por B, não estará disponível quando da sua necessidade. Este é o mesmo problema clássico de *data hazard* resolvido em arquiteturas RISC pela unidade de *forwarding*. A diferença neste caso é que se tratam de duas operações dependentes sendo computadas em diferentes *datapaths*. A figura 2.6 ilustra os pontos no tempo onde o registrador definido por A estará no banco centralizado de registradores e o ponto anterior onde a operação B necessita deste dado.

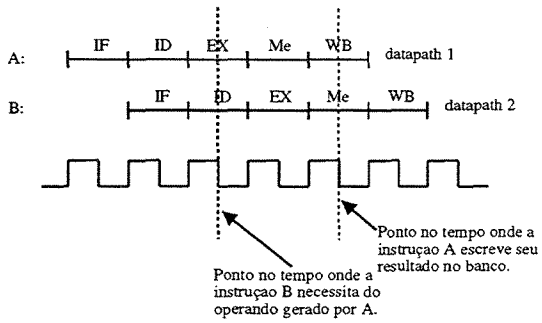


Figura 2.6: Diagrama de tempo para as instruções A e B sendo executadas em diferentes *datapaths* sincronizados.

Nota-se que a instrução B necessita do dado definido por A dois ciclos de relógio antes deste operando ser escrito no banco de registradores. Isso faz com que a seqüencia das instruções possua duas bolhas no *pipeline* entre as operações A e B, aumentando o número de ciclos necessários para o término da execução. A tabela 2.2 ilusta este fato.

Abnous [2] sugere *hardware* extra para provocar *stalls* ao invés de instruções de NOP, o que degrada ainda mais o desempenho pois todos os *datapaths* são paralizados nes-

FU 1	FU 2
A	
NOP	
NOP	
	B

Tabela 2.2: Escalonamento resultante.

ta abordagem. Os *stalls* devem ser propagados para todos os *datapaths* para evitar a desincronização dos mesmos.

Se, entretanto, considerarmos uma arquitetura particionada, podemos estabelecer uma comunicação completa entre todas as unidades funcionais de um mesmo *cluster*, aumentando assim a possibilidade de redução de *data hazards*. Claramente, o número de unidades funcionais de uma partição não deve exceder um certo limite máximo a ponto de comprometer o caminho crítico do circuito pela adição dos barramentos de *bypassing*.

Um outro cenário a ser considerado aparece quando duas operações dependentes são escalonadas em unidades funcionais não comunicantes e pertencentes a diferentes *clusters*. Neste caso, uma instrução de cópia é necessária para a transferência dos operandos e deve ser inserida entre as duas operações. Deste modo, a tentativa de redução do número de instruções de cópia em uma arquitetura VLIW particionada é similar à tentativa de redução no número de *stalls* realizada nas arquiteturas não-particionadas. A figura 2.7 e a tabela 2.3 mostram o comportamento das instruções A e B sendo executadas em *clusters* distintos de uma arquitetura particionada.

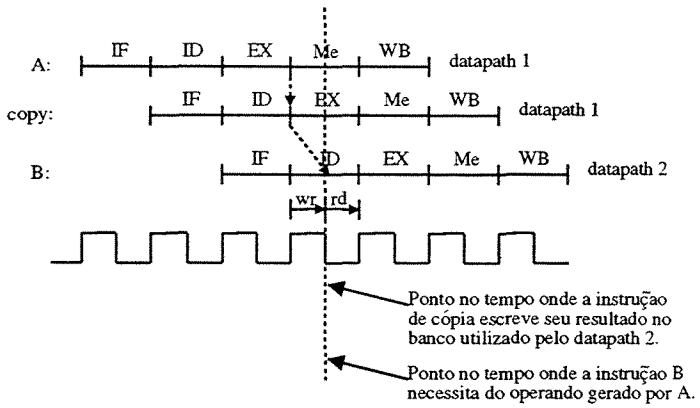


Figura 2.7: Diagrama de tempo para as instruções A, B e C (cópia) sendo executadas em diferentes *datapaths* sincronizados.

FU 1	FU 2
A	
COPY	
	B

Tabela 2.3: Escalonamento resultante.

Note que a instrução de cópia é executada no mesmo *datapath* da instrução A. Neste caso, *bypassing* local é utilizado para prover o operando-fonte da instrução de cópia, ainda na segunda metade do seu ciclo de decodificação. Na primeira metade do ciclo de execução, a instrução de cópia transfere o operando lido para o banco de registradores destino. Isso significa que este operando estará pronto para ser lido pela instrução B no momento correto, ou seja, na segunda metade do ciclo de decodificação desta instrução.

Entretanto, para que este tipo de cópia seja possível, a modificação na arquitetura mostrada na figura 2.8 deve ser considerada. A explicação para isso vem do fato de que, devido ao roteamento utilizado na arquitetura proposta por Dutt [8], o operando-fonte da instrução de cópia deve vir obrigatoriamente do banco de registradores. Na arquitetura sugerida nesta dissertação, o roteamento utilizado para as cópias sai do *pipeline* de origem no meio do estágio de execução. Pode-se afirmar que em termos de ciclos nenhuma das duas arquiteturas traz desvantagem em relação à outra. Na arquitetura proposta por Dutt define-se que, no momento da cópia, a unidade de execução ligada ao banco de origem deve ficar sem operar durante o tempo necessário para a movimentação do dado (1 ciclo). Aqui, a unidade funcional é também responsável pela *execução* da instrução de cópia, portanto continua "trabalhando" no ciclo em que ela ocorre. Note-se que ao final da primeira metade do ciclo de execução o dado já estará no banco destino. Isto é viável praticamente uma vez que o operando-fonte da cópia foi lido na segunda metade do estágio de decodificação. Em resumo, a modificação sugerida possibilita que o operando-fonte da instrução de cópia seja proveniente tanto do banco de registradores (operações dependentes distantes) quanto de uma linha de *forwarding* local (operações dependentes próximas).

A figura 2.9 ilustra com mais detalhes a diferença entre os dois modelos de arquitetura: o proposto por Dutt (fig. 2.9.a) e o sugerido nesta dissertação (fig. 2.9.b).

A análise anterior mostra que a inserção da instrução de cópia não afeta o comprimento final do escalonamento quando comparada à inserção de NOPs resultante da falta de *bypassing* entre unidades de execução de uma arquitetura com um só banco de registradores. Ao contrário, ao invés das duas instruções de NOP do primeiro caso, apenas uma instrução de cópia necessita ser gerada.

Imagine, entretanto, se pudéssemos prover "linhas" de *forwarding* entre as duas uni-

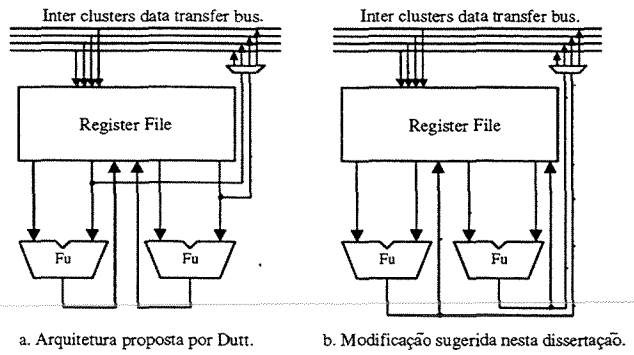


Figura 2.8: Diferenças entre a arquitetura proposta por Dutt e a adotada nesta dissertação.

dades em que as operações A e B foram escalonadas, mesmo sendo elas pertencentes a *clusters* diferentes. Isso eliminaria a necessidade da própria instrução de cópia, caso as instruções A e B estivessem próximas o suficiente para usufruírem desta capacidade.

Esta é uma das idéias em que esta dissertação se baseia no sentido de reduzir o número de instruções de cópias eventualmente necessárias em arquiteturas particionadas. Esta abordagem leva a uma especialização do processador, caracterizando um mapeamento de uma única aplicação em uma arquitetura dedicada. Geralmente o esforço desta especialização tem como alvo uma classe de aplicações, e não somente uma delas. Por outro lado, esta abordagem pode tornar eficiente uma arquitetura VLIW com vários *clusters* sem a desvantagem de um intenso roteamento devido ao *bypassing* entre os mesmos. Limitando-se o volume de roteamento permitido para a arquitetura, podemos computar, num estágio pós-escalonamento, quais foram as unidades que mais se comunicaram devido a operações dependentes terem sido escalonadas nas mesmas. Deste modo, *bypassing* pode ser adicionado somente entre as unidades mais comunicantes reduzindo o número de ciclos perdidos com NOPs e cópias. A figura 2.10 ilustra como o *bypassing* é organizado nas arquiteturas RISC e como pode ser organizado neste modelo de arquitetura. Vale ressaltar que, em toda a extensão deste trabalho, *bypasing* local, ou seja, entre os diferentes estágios de um mesmo *pipeline*, estará sempre sendo considerada como *default*. Caso contrário, teríamos *data hazards* mesmo que duas operações dependentes fossem escalonadas na *mesma* unidade funcional.

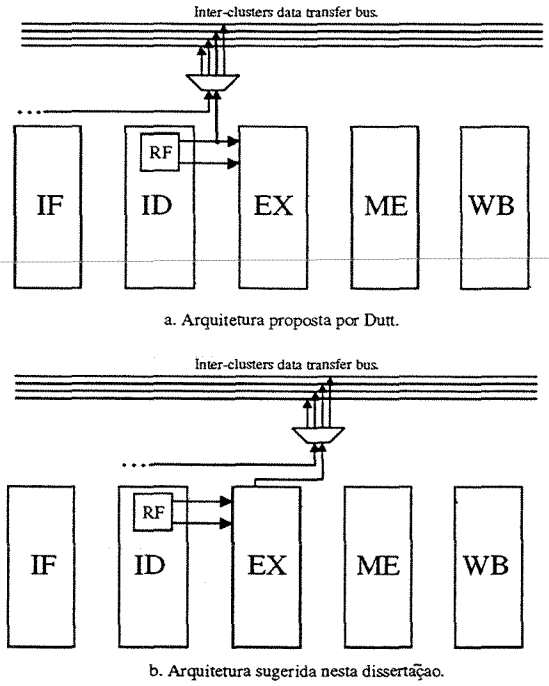


Figura 2.9: Diferença entre a arquitetura proposta por Dutt e a sugerida nesta dissertação.

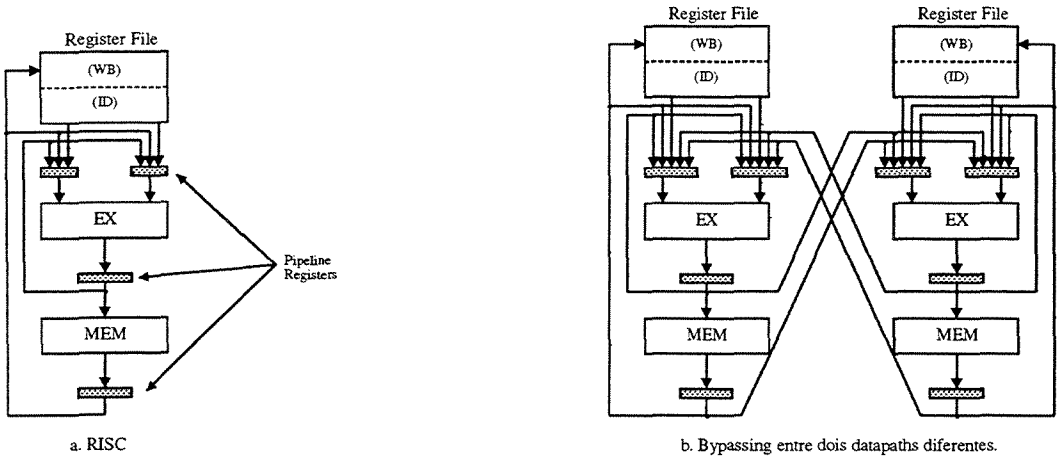


Figura 2.10: Organização do *bypassing*.

Capítulo 3

Em Busca de Operações Paralelizáveis

Conforme citado no capítulo 2, a principal característica dos processadores VLIW é a execução paralela de um conjunto de operações independentes empacotadas sob a forma de uma instrução longa. Essas instruções longas, por sua vez, são geradas pelo compilador que deve garimpar o código em busca de operações independentes e paralelizáveis. Assim, uma característica fundamental para um compilador que deseja gerar código para uma arquitetura paralela é a vasta utilização de técnicas que resultem em um aumento significativo do volume de paralelismo encontrado em programas escritos seqüencialmente. O tamanho original dos blocos básicos destes programas é de aproximadamente cinco instruções, limitando drasticamente as possibilidades de paralelismo. Assim sendo, para que possamos utilizar eficientemente os recursos de *hardware* disponíveis em um processador VLIW, técnicas de busca por um aumento do paralelismo além dos limites de simples blocos básicos devem obrigatoriamente fazer parte do processo de compilação. Isso também é valido para áreas de pesquisa como *High-Level Synthesis* [60, 26].

Algumas das técnicas mais utilizadas para o aumento do volume de paralelismo em programas originalmente seqüenciais foram comentadas na parte introdutória desta dissertação. Um aprofundamento destas estratégias é apresentado neste capítulo, tentando-se realizar uma interligação entre esses algoritmos e o contexto de arquiteturas particionadas abordado neste trabalho. Ainda, apresentamos a implementação do *Grafo de Dependências do Programa* realizada que serve como base para uma análise do potencial paralelismo encontrado nos *benchmarks* mais comumente empregados em aplicações embarcadas. Este grafo tem a característica de também explorar o paralelismo além dos limites dos blocos básicos, aglutinando vários blocos potencialmente paralelizáveis sob a forma de regiões equivalentes.

3.1 Loop Unrolling

Paralelismo existente em laços pode ser inibido pelas dependências das instruções às variáveis de índice usadas para acesso de vetores. Essas variáveis são geralmente incrementadas a cada iteração, assim como as variáveis de indução [4], criando dependências de dados indesejáveis entre iterações. Entretanto, essas dependências são apenas o resultado da maneira com que o código é gerado para processadores escalares, não sendo parte inerente da semântica dos laços.

Assim, paralelismo no nível de instrução em laços pode ser buscado além das fronteiras de uma iteração através de um método denominado *loop unrolling*. Este método consiste em replicar o corpo do laço várias vezes de forma que o laço resultante contenha várias iterações do laço original. Como conseqüência, um volume maior de paralelismo é exposto. Todavia, esse aumento somente será significativo se o número de dependências entre iterações (*loop-carried dependences*) for pequeno. O número de vezes que o laço é desenrolado é denominado fator de *unrolling*.

Se o *loop-unrolling* é realizado em alto-nível, os acessos à memória presentes no corpo do laço devem ter seus índices de endereçamento revistos a fim de referenciarem a posição correta no *array*. Mesmo no nível de linguagem de máquina, os endereços de memória das operações *load* e *store* devem ser redirecionados para cada uma das vezes em que o corpo do laço for replicado. As figuras 3.1, 3.2 e 3.3 mostram um exemplo de aplicação de *loop-unrolling* destacando o cálculo correto dos endereços.

```
(1)     #define K 4
(2)     y[i] = 0;
(3)     for(j=0; j<K; j++) {
(4)             y[i] = y[i] + x[i+K-j-1] * (j+1);
(5)     }
```

Figura 3.1: Código-fonte de um laço a ser desenrolado.

Para a maioria dos programas não-numéricos, *loop-unrolling* apresenta apenas uma pequena contribuição. Estes programas apresentam laços com uma quantidade razoável de dependências de dados e controle, além de iterarem um pequeno número de vezes [64]. Por outro lado, considerando-se uma máquina com suporte a execução predicada, o número elevado de instruções de controle presentes no corpo do laço poderiam ser mascaradas pela técnica de *if-conversion* [44] e o impacto da técnica de *loop-unrolling* seria mais significativo.

Programas numéricos que computam operações sobre matrizes são particularmente

```
(1)         y[i] = 0;
(2)         j = 0;
(3)         y[i] = y[i] + x[i + 4 - j - 1] * (j + 1);
(4)       unroll_1:
(5)         j = j + 1;
(6)         y[i] = y[i] + x[i + 4 - j - 1] * (j + 1);
(7)       unroll_2:
(8)         j = j + 1;
(9)         y[i] = y[i] + x[i + 4 - j - 1] * (j + 1);
(10)      unroll_3:
(11)        j = j + 1;
(12)        y[i] = y[i] + x[i + 4 - j - 1] * (j + 1);
(13)      unroll_4:
(14)        j = j + 1;
(15)      brk;
```

Figura 3.2: Corpo do laço mostrado na figura 3.1 repetido quatro vezes.

```
(1)         y[i] = 0;
(2)         y[i] = y[i] + x[i + 3];
(3)         y[i] = y[i] + x[i + 2] * 2;
(4)         y[i] = y[i] + x[i + 1] * 3;
(5)         y[i] = y[i] + x[i] * 4;
```

Figura 3.3: Resultado final após a fase de *cleanup*.

beneficiados pela estratégia de *loop unrolling*.

As principais desvantagens da técnica de *loop-unrolling* são o aumento no tamanho de código e o aumento na pressão por registradores[1] [4, 44]. Geralmente são esses os fatores que limitam o número de vezes que um laço é desenrolado.

De fato, uma característica desta técnica é a dificuldade de obter-se o fator de unrolling ideal para cada laço. Segundo Hwu, "...it is hard to set the number of unrolls intelligently..." [32]. O compilador IMPACT provê uma função denominada *find_num_unroll* que tenta, por meio de heurísticas, calcular o número de vezes que o laço passado como parâmetro deve ser desenrolado[2]. Em arquiteturas VLIW particionadas, todavia, uma

[1] Do inglês, *register pressure*

[2] Comentário inserido no arquivo fonte *l_loop_unroll.c* do compilador IMPACT: *First look for an*

boa métrica é desenrolar o laço um número de vezes igual ao número de *clusters* da arquitetura. Desta forma, cada iteração tende naturalmente a ser atribuída a um cluster distinto [58, 57, 59].

3.1.1 Software Pipelining

Software pipelining [38, 28] é uma técnica de otimização que tem como finalidade melhorar o desempenho da execução de laços em qualquer sistema que permita paralelismo no nível de instrução (ILP), incluindo arquiteturas VLIW e super-escalares. Esta técnica permite que partes distintas de várias iterações de um laço possam ser processadas simultaneamente, tirando proveito do paralelismo disponível no corpo do laço. Se A,B e C são partes de um laço, com dependências entre elas, a idéia básica desta técnica pode ser resumida na seguinte expressão:

$$(ABC)^n = A(BCA')^{n-1}BC$$

Em outras palavras, tenta-se definir um prólogo, um epílogo e um corpo modificado para o laço, a fim de se encontrar um núcleo onde a ordem das operações contidas maximize o paralelismo. A procura pelo prólogo, epílogo e núcleo se dá através de uma sobreposição de várias iterações sucessivas do laço, porém sem desenrolar o mesmo. Um exemplo da aplicação de *software pipelining* é mostrado a seguir. A figura 3.4 mostra um grafo que representa o corpo de um laço sem instruções condicionais. As operações são representadas pelos nós do grafo e as dependências entre operações pelas arestas. Dependências entre iterações são mostradas em linhas pontilhadas.

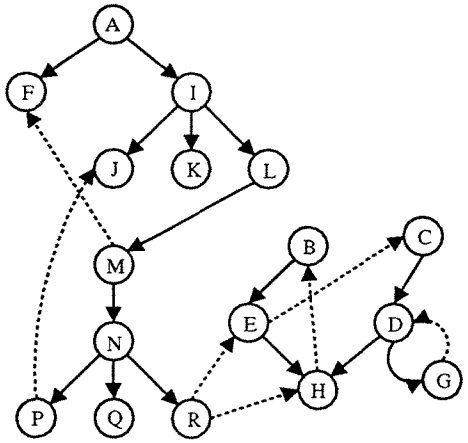


Figura 3.4: Grafo representando o corpo de um laço.

attribute in which a divine force has told us how many times to unroll the loop :-).

Um escalonamento *greedy* deste grafo, sem a utilização da técnica de *software pipelining* é visto na tabela 3.5. Esta tabela mostra a distribuição das operações em cada iteração do laço juntamente com o ciclo de relógio associado a cada uma.

	Iterations				
time	1	2	3	4	5
1	ABC	A	A	A	A
2	DEFI	I	I	I	I
3	GHJKL	CKL	KL	KL	KL
4	M	BDM	M	M	M
5	N	EFGN	FN	FN	FN
6	PQR	PQR	CPQR	PQR	PQR
7		HJ	DJ	J	J
8			BG	-	-
9			E	-	-
10			H	C	-
11				BD	-
12				EG	-
13				H	C
14					BD
15					EG
16					H

Figura 3.5: O código após cinco etapas.

As operações são distribuídas em cada iteração conforme as dependências de dados entre elas vão sendo resolvidas. Assim, no primeiro ciclo da primeira iteração as operações A,B e C (ver figura 3.4) são as únicas operações independentes e portanto são alocadas neste ciclo. No ciclo seguinte, ainda na mesma iteração, as operações D, E, F e I tornam-se as operações prontas e portanto são escalonadas. Assim ocorre para todos os outros nós do grafo até que todas as operações tenham sido alocadas para a primeira iteração.

Para a segunda iteração, a única operação pronta no primeiro ciclo é a operação A, visto que as *loop-carried dependences* entre as operações $H \rightarrow B$ e $E \rightarrow C$ não possibilitam a alocação de B e C neste ciclo (ver figura 3.4 e tabela 3.5). Assim, verificando-se as dependências do grafo e a distribuição das operações na iteração anterior, sucessivamente vai-se construindo a tabela mostrada na figura 3.5.

Note que a partir do ciclo 4 começam a aparecer lacunas entre as operações J e C, os quais tendem a aumentar a cada nova iteração. Este fato não possibilita encontrarmos um núcleo comum ao laço sendo analisado a não ser que, ao invés de adicionarmos espaços, baixemos a coluna corrente até todos eles serem preenchidos. Esta é a metodologia que caracteriza a técnica de *software pipelining*. A figura 3.6 mostra o resultado final da aplicação desta técnica enfatizando o núcleo encontrado para o laço.

Observe que a técnica de *software pipelining* é aplicada apenas para se encontrar um núcleo onde todas as operações do laço estejam incluídas. *Loop unrolling* pode ser combinado com *software pipelining* a fim de aumentar o volume de paralelismo do laço. Ainda, *software pipelining* não realiza o escalonamento das instruções propriamente dito. Um algoritmo escalonador como *list scheduling* deve ser aplicado ao prólogo, núcleo e

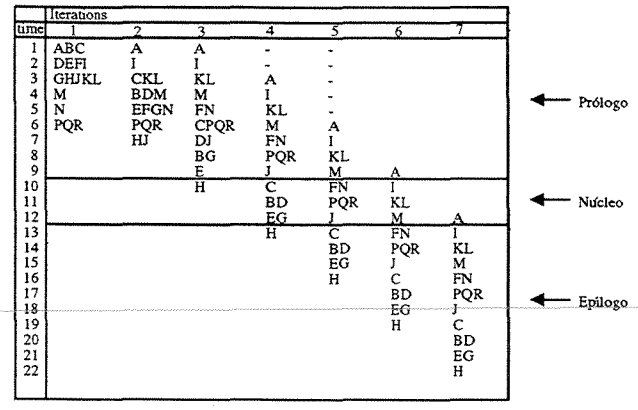


Figura 3.6: *Software Pipelining* sendo aplicado ao laço da figura 3.4.

epílogo a fim de associar operações à unidades funcionais da arquitetura alvo para a qual a aplicação está sendo compilada.

3.2 Trace Scheduling

Trace scheduling [22, 24, 28] identifica os caminhos (traços) mais freqüentemente executados de cada procedimento e escalona as instruções nestes traços como se as mesmas pertencessem a um grande bloco básico. Diferentemente de *software pipelining*, os resultados parciais da técnica de *trace scheduling*, assim como o resultado final, já é o código compactado e escalonado para a arquitetura alvo. A figura 3.7 mostra os passos necessários para a formação dos traços de execução.

Podem existir saltos condicionais saindo do interior do traço (saídas laterais) e transições de outros traços para um determinado traço (entradas laterais). Todavia, as instruções são escalonadas em cada traço sem levar em conta essas transições de controle. Após o escalonamento, instruções de compensação devem ser eventualmente adicionadas a alguns blocos, a fim de manter a semântica original do procedimento. De um certo ponto de vista, esta técnica pode ser vista como uma estratégia que se baseia em especulação e predição de saltos.

Informação dinâmica (predição de saltos) é usada em tempo de compilação para selecionar os traços com maior probabilidade de execução. Na parte *b* da figura 3.7 pode-se notar um traço sendo extraído do grafo de fluxo de controle. Vale ressaltar que a região do CFG sendo analisada não pode conter laços.

Assim os traços vão sendo identificados e escalonados, inserindo-se instruções de compensação quando necessárias.

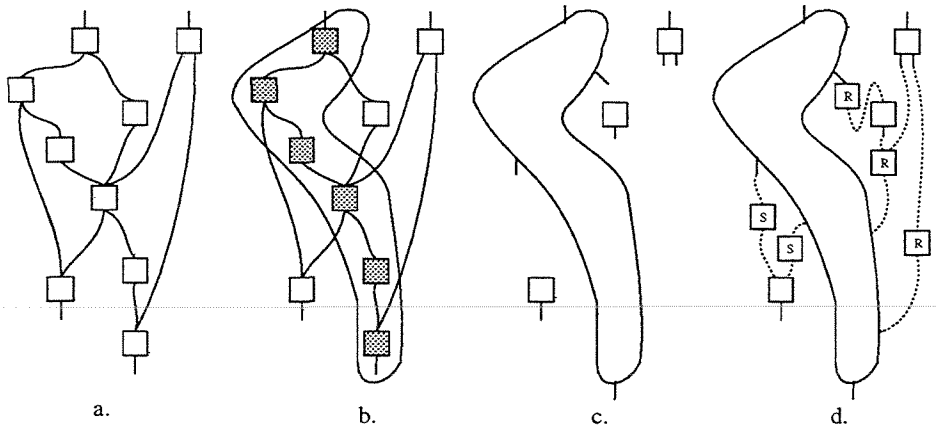


Figura 3.7: Trace scheduling.

3.3 Formação de Superblocos

A formação de um superbloco [33] equivale a formação de um traço (sub-seção 3.2) que não permite a existência de entradas laterais. O fluxo de controle pode somente entrar pelo topo do superbloco e sair por um ou mais pontos de saída. A formação dos superblocos se dá em duas etapas. A primeira delas consiste na formação convencional dos traços de execução mais freqüentes através de informações de *profiling*. O próximo passo é a aplicação de uma técnica chamada *tail duplication*, que elimina as entradas laterais do traço pela replicação do chamado "bloco de saída" [33] . Este bloco é copiado e as entradas laterais são redirecionadas para que o superbloco seja formado de acordo com a sua definição. Um exemplo de formação de superblocos é mostrada na figura 3.8. A figura 3.8.a mostra um grafo de fluxo de controle ponderado representando um segmento de código de um laço. Os nós correspondem à blocos básicos e as arestas representam as possíveis transferências de controle entre os blocos. Cada bloco básico tem associado a respectiva freqüência de execução. Da mesma forma, cada aresta tem associado um valor que indica a freqüência na qual as respectivas transferências de controle são invocadas.

No exemplo, o caminho de execução mais freqüentemente executado corresponde à seqüência <A,B,E,F>. No total, existem três traços, <A,B,E,F> , <C> e <D>. Após a seleção dos traços, cada um deles é convertido para um superbloco. Na figura 3.8.a observa-se que existem dois fluxos de controle penetrando o traço <A,B,E,F> aterrisando no bloco básico F. Este bloco básico duplicado forma um novo superbloco que é anexado ao final da função. O resultado é mostrado na figura 3.8.b. Nesta figura vemos que as entradas laterais do traço <A,B,E,F> foram removidas, transformando-o em um superbloco.

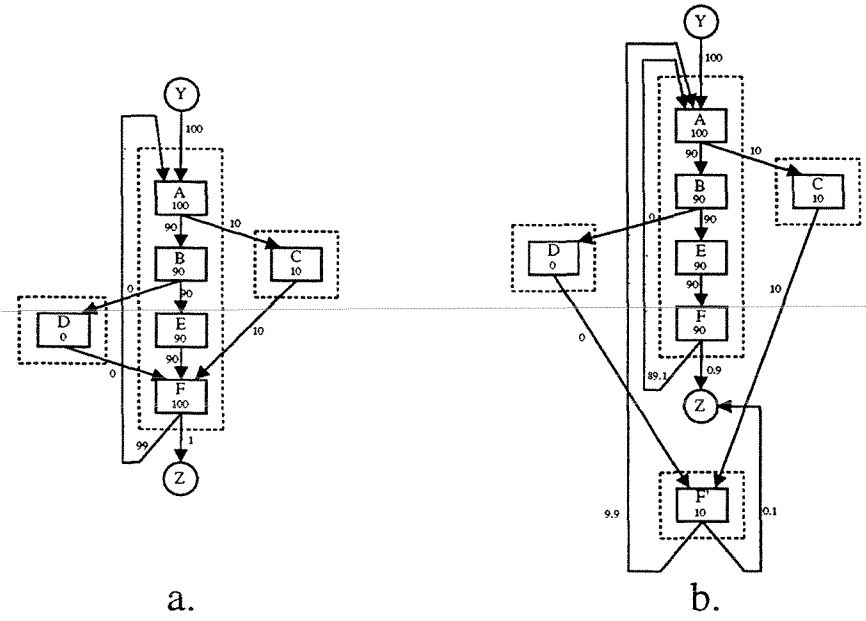


Figura 3.8: Formação de superblocos.

Superblocos são estruturas similares aos chamados blocos básicos estendidos [4]. Um bloco básico estendido é definido como uma seqüência de blocos básicos $B_1...B_k$ tal que para $1 \leq i \leq k$, B_i é o único predecessor de B_{i+1} e B_1 não possui somente um predecessor [4]. A diferença entre os superblocos e os blocos básicos estendidos está na maneira com que os mesmos são formados. A formação dos superblocos é guiada fundamentalmente por informações de *profile* e retirada das entradas laterais dos traços com o objetivo de aumentar o tamanho daqueles. Além disso, um superbloco pode possuir um único predecessor, assim como os blocos básicos estendidos.

A técnica de *loop-unrolling* pode ainda ser aplicada a um superbloco quando o mesmo formar o corpo de um laço. Para desenrolar N vezes um laço com esta característica, *N-1* cópias do superbloco são anexadas ao superbloco original.

3.4 Formação de Hiperblocos

Os superblocos vistos na seção anterior não consideram a possibilidade de predicação de instruções, um método eficiente para lidar com saltos condicionais. Uma estrutura similar ao superbloco, denominada hiperbloco [43], mescla as características do superbloco com a utilização de predicação para tornar o escalonamento de instruções mais eficiente em laços com condicionais. Um hiperbloco é formado por um conjunto de blocos básicos que

apresentam instruções predicadas. O fluxo de controle pode somente entrar pelo topo do hiperbloco, podendo porém sair por um ou mais pontos. Um bloco básico único é designado como entrada do hiperbloco.

Para lidar com predicação, hiperblocos são formados utilizando-se uma versão modificada da técnica *if-conversion*. Ainda, execução especulativa é possível através de uma propagação de predicados no interior do hiperbloco. No contexto de predicação de instruções, execução especulativa significa executar uma operação antes de saber o valor de seu predicado.

Assim, o primeiro passo na formação de um hiperbloco é a decisão de quais blocos básicos de uma dada região do programa farão parte do hiperbloco. Tipicamente, uma região a ser considerada para a formação de um hiperbloco são os blocos básicos pertencentes aos laços mais internos do programa. Blocos básicos são inseridos no hiperbloco de acordo com sua freqüência de execução, tamanho e características das instruções. Após escolhidos quais blocos básicos formarão o hiperbloco, técnicas convencionais de *if-conversion* são aplicadas às instruções destes blocos para agrupar os diferentes fluxos de execução em um único fluxo.

Um exemplo de formação de hiperbloco é visto na figura 3.9. A figura 3.9.a mostra um grafo de fluxo de controle típico com informações de *profile* inseridas nos blocos e arestas. Vemos que a técnica de *tail duplication*, considerada nos superblocos, também é aplicada aqui. A diferença básica de hiperblocos e superblocos é notada através da inclusão do bloco básico C no hiperbloco <ABCDE>. Sem a possibilidade de predicação (que é o caso dos superblocos) essa inclusão se torna impossível.

3.5 Grafo de Dependências do Programa

Uma outra metodologia utilizada na busca por operações paralelizáveis de um programa baseia-se em uma estrutura de dados denominada Grafo de Dependências do Programa (PDG)[3] [21]. Esta representação tem por característica explicitar todas as dependências de controle e de dados existentes entre as instruções de um procedimento. Através da análise destas dependências pode-se quantificar o volume de paralelismo disponível na aplicação e, através destes números, determinar se um esforço no escalonamento das instruções possivelmente usando somente o PDG [30, 6] será ou não recompensado. Transformações que aumentem o volume de paralelismo podem também ser aplicadas sobre o PDG para que uma quantidade maior de paralelismo seja atingido [30].

Conforme citado no início deste capítulo, a busca por operações paralelizáveis realizada somente no escopo de simples blocos básicos não resulta em ganhos significativos. Assim,

[3]Do inglês, *Program Dependence Graph*

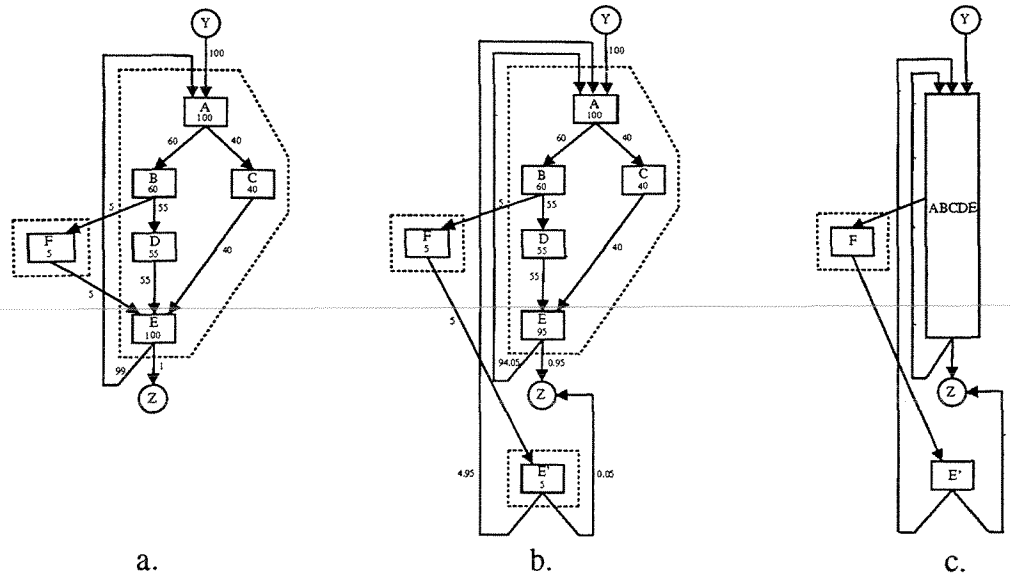


Figura 3.9: Formação de Hiperblocos.

as técnicas vistas nas últimas seções garimpam o código atrás de operações independentes em um escopo que vai além dos limites de um único bloco básico, valendo-se para isso de diferentes metodologias. A utilização do PDG também tem esse objetivo, ou seja, explorar o paralelismo entre instruções além dos limites de blocos básicos individuais. No PDG, *nós de região* agrupam blocos básicos que são independentes sob o ponto de vista do controle. Em outras palavras, instruções pertencentes a esses blocos *controle-equivalentes* são fortes candidatas a uma execução paralela, caso as dependências de dados não limitem este paralelismo. Isso significa que o PDG também "enxerga" além dos limites dos blocos básicos individuais. Todavia, a questão aqui é se os programas apresentam um volume de paralelismo razoável quando verificarmos a quantidade de blocos básicos *controle-equivalentes* em cada nó de região e a quantidade total de operações desses blocos.

Formalmente, o Grafo de Dependências do Programa representa um programa sob a forma de um grafo no qual os vértices são blocos básicos e as arestas representam as dependências de controle e de dados entre os vértices. As dependências, portanto, resultam de dois efeitos separados. Primeiro, uma dependência existe entre duas sentenças quando uma variável que aparece em uma delas tem um valor incorreto se a ordem de execução das duas sentenças é invertida. Dependências deste tipo são conhecidas como dependências de dados. Através da análise de fluxo de dados denominada "definições alcançáveis" (*reaching definitions*) [4] pode-se extrair todas as informações necessárias para o cálculo das dependências de dados de um programa. *Reaching definitions* á aplicada

sobre o grafo de fluxo de controle do procedimento.

Segundo, uma dependência existe entre uma sentença e o predicado cujo valor controla a execução desta sentença. Isto é, a decisão tomada no final do bloco básico X determina se o bloco básico Y será ou não executado. Neste caso, diz-se que o bloco Y é dependente de controle do bloco básico X sob a aresta na qual o caminho determina a execução de Y. As dependências de controle que existem entre os blocos básicos são ditadas pela natureza dos diferentes desvios encontrados no programa.

Com relação aos vértices, além dos blocos básicos outro tipo de vértice compõe o PDG. Estes vértices são denominados *Nós de Região*, e têm a finalidade de agrupar todos os blocos básicos que possuem as mesmas condições de controle sob um mesmo vértice comum.

Esta forma de representação hierárquica é extremamente útil numa posterior busca por operações paralelizáveis, visto que todos os "filhos" de um mesmo nó de região são fortes candidatos a execução paralela. Desta maneira, o PDG identifica os blocos básicos potencialmente paralelizáveis, não importando o quão longe eles possam estar um do outro no CFG. Os blocos básicos *controle-equivalentes* podem ser vistos como um único macro-bloco básico, aumentando desta forma o escopo do escalonamento.

3.5.1 Construção do PDG

O PDG é construído a partir do grafo de fluxo de controle e das dependências de dados entre operações. A figura 3.10 apresenta um grafo de fluxo de controle típico de uma subrotina.

Os passos para a construção do PDG seguem o seguinte pseudo-algoritmo:

1. Construa o grafo de fluxo de controle (CFG) [4] da rotina.

2. Construa a árvore de pós-dominadores [4] da rotina. A computação dos pós-dominadores no grafo de fluxo de controle é equivalente ao cálculo dos dominadores no grafo de fluxo de controle invertido. Dominadores no CFG invertido podem ser computados rapidamente usando o algoritmo de Lengauer e Tarjan [41]. [4] .

3. Dados o CFG e a árvore de pós-dominadores do mesmo, construir o conjunto "S", formado por todas as arestas do CFG nas quais o bloco básico destino da aresta *não* pós-domina o bloco básico origem (arestas "A $\rightarrow$ B" tal que B não pós-domina A).

4. Finalmente, a determinação das dependências de controle do procedimento provém da investigação de todas as arestas do conjunto "S": para cada aresta deste conjunto, executar os seguintes passos:

[4] O algoritmo de Lengauer e Tarjan é o método implementado pelo compilador GCC versão 3.0.

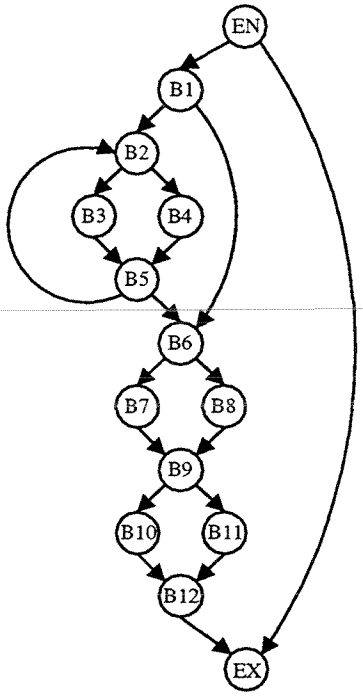


Figura 3.10: Grafo de Fluxo de Controle de uma subrotina.

- Localizar, na árvore de pós-dominadores, o ancestral-comum L dos blocos básicos origem e destino (A e B) da aresta sendo analisada .
- Todos os nós da árvore de pós-dominadores no caminho de L até B, incluindo B mas não L, são dependentes de controle de A. Caso L seja o próprio A, incluir L como dependente de controle de A. Em outras palavras, A passa a ser dependente de controle dele próprio (laços).

Analisando este procedimento calmamente, pode-se concluir que, dados A e B, o efeito desejado é atingido percorrendo-se todos os nós desde B até o ancestral-comum L, marcando todos os nós neste caminho como dependentes de controle de A.

Com as informações sobre dependências de controle retiradas com o uso do algoritmo acima, o PDG é trivialmente construído hierarquizando-se as informações obtidas sob a forma de um grafo.

O próximo passo na construção do subgrafo de dependências de controle é a adição dos chamados "nós de região". Estes nós especiais, que não representam blocos básicos, tem a finalidade de agrupar sob uma mesma hierarquia os blocos básicos que possuem as mesmas dependências de controle.

A figura 3.11 mostra o PDG do programa representado na figura 3.10, diferenciando

os nós que representam blocos básicos dos nós que representam regiões. Nós de região estão hachurados na figura. As dependências de dados foram omitidas da figura em prol de uma maior clareza. Para mostrá-las, teríamos que expandir os blocos-básicos no nível de operações individuais e desenhar arestas entre operações dependentes, sejam elas intra ou inter blocos básicos.

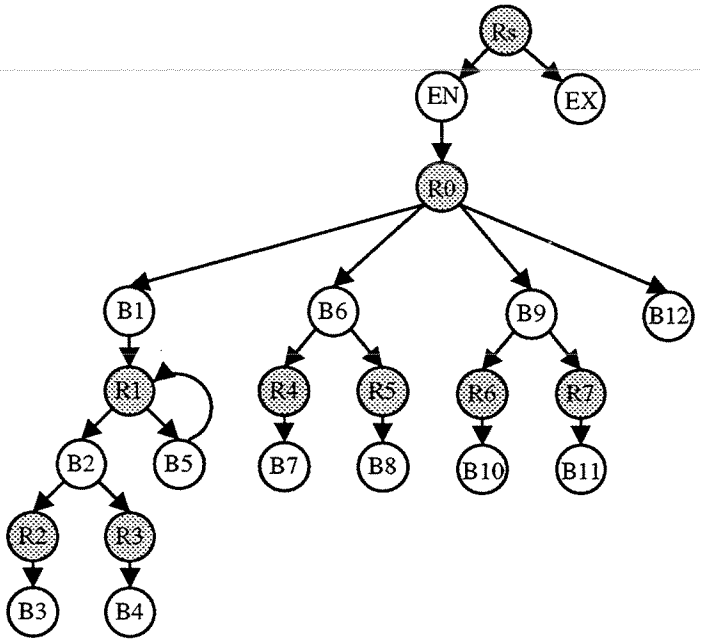


Figura 3.11: PDG do programa representado na figura 3.10.

Uma observação importante sobre o PDG é que ele conduz a um meio natural para a tarefa de busca por operações paralelizáveis. Outra característica peculiar a este grafo é que, para programas estruturados, o PDG é semelhante a árvore de sintaxe abstrata do programa. Isto é devido ao fato de que as informações de fluxo de controle de um programa são precisamente capturadas pela sintaxe do mesmo. De fato, muitos compiladores paralelos para linguagens de programação estruturadas dependem da sintaxe do programa para prover informações de dependências de controle. Entretanto, a sintaxe de um programa não pode fornecer informações de dependências de controle no caso de programas não estruturados. Neste caso, abordagens mais genéricas como o PDG devem ser utilizadas.

3.6 Resultados Experimentais

Com o objetivo de analisar o volume de paralelismo disponível em aplicações embarcadas, foi realizada uma implementação do PDG no código-fonte do compilador GCC [29]. Até onde sabemos, esta é a primeira implementação do PDG neste compilador. GCC é um compilador que implementa várias técnicas de otimização a fim de prover alta qualidade ao código-objeto gerado. Para isso, otimizações como eliminação de subexpressões comuns [4], propagação de expressões constantes [4], eliminação de comandos de cópia [4], remoção de código invariante em laços [4], eliminação de código inútil [4], etc. são vastamente utilizadas.

A metodologia adotada para a implementação foi gerar um *Cross-compiler* para o processador MIPS a partir de uma máquina CISC (Pentium) rodando o sistema operacional Linux. O processador-alvo (MIPS) foi escolhido por se tratar de um processador RISC padrão e também por ser usado em certos nichos de sistemas embutidos.

Para cada procedimento da aplicação o compilador modificado constrói o CFG (já presente no código original) e o PDG. A partir deste ponto, o compilador gera estatísticas como número total de blocos básicos, número de nós de região, número de blocos básicos *controle-equivalente* por nó de região e número de instruções potencialmente paralelizáveis por nó de região. Com estes dados pode-se analisar o volume médio de paralelismo disponível em cada uma das aplicações testadas. Como aplicações-alvo foram escolhidos os *benchmarks* SPECINT95, SPECFP95 e alguns outros programas aos quais denominaremos *Miscelaneous*. O conjunto *Miscelaneous* também é formado por programas comumente encontrados na literatura. Por se tratarem de *benchmarks* bastante conhecidos, a descrição dos mesmos foi omitida desta dissertação.

3.6.1 Análise dos Resultados

A tabela 3.1 resume alguns dos resultados obtidos a partir da implementação do PDG no compilador GCC.

Nesta tabela 3.1, a coluna *BBs / Nó de Região* mostra o número médio de blocos básicos em cada nó de região do PDG para os diversos programas testados. Essa medida nos dá uma "fotografia" do volume de paralelismo capturada tendo-se como fator limitante as condições de controle da aplicação.

A coluna *Inst. / Nó de Região* traz a medida do número de instruções potencialmente paralelizáveis em cada nó de região. Além de contar-se o número médio de blocos básicos em cada nó de região também computa-se o número médio de *instruções* sujeitas ao mesmo nó de região.

Finalmente, a coluna *Inst. / BB* mostra o número médio de instruções em cada bloco básico das aplicações.

Benchmark	Programa	# de Subrotinas	BBs por Nó de região	Inst. por Nó de Região	Inst. por BB
SPECInt 95	099.go	372	1.2	6.99	5.82
	124.m88ksim	252	1.1	5.69	5.17
	126.gcc	1762	0.98	4.47	4.52
	129.compress	24	1.48	8.44	5.70
	130.li	357	1.12	5.82	5.17
	132.ijpeg	467	1.3	7.52	5.78
	134.perl	275	1.14	5.51	4.81
	147.vortex	919	1.11	6.94	6.2
SPECFp 95	101.tomcatv	1	1.64	13.8	8.42
	102.swim	7	2.32	31.5	13.52
	103.su2cor	38	1.69	21.6	12.8
	104.hydro2d	43	1.8	14.5	8.04
	107.mgrid	13	1.94	23.5	12.12
	110.applu	17	2.06	32.4	15.74
	125.turb3d	11	1.7	21.5	12.62
	141.apsi	97	1.67	22.8	13.67
	145.fpppp	38	1.33	32.2	24.09
	146.wave5	111	1.67	16.9	10.1
Misc.	gsm	57	1.3	8.67	6.67
	mpeg2dec	114	1.32	7.12	5.36
	mpeg2enc	95	1.36	8.04	5.87
	paraffins	10	1.71	7.34	4.28
	kalman	76	1.65	12.35	7.45
	fir	4	1.45	8.9	6.09
	dag	1	1.33	5.11	3.83
	eight	1	1.5	6.5	4.33

Tabela 3.1: Resultados obtidos a partir da implementação do PDG no compilador GCC.

Através dos resultados apresentados podemos notar claramente uma certa superioridade de paralelismo por parte dos programas retirados do *benchmark* SPECFP95. Dois fatores contribuem para isso: essas aplicações possuem menos condições de controle, observável pelo maior número de blocos básicos por nó de região, e possuem um maior número médio de instruções em cada bloco básico. Os dois fatores combinados resultam em um maior volume líquido de paralelismo em potencial. Outra conclusão importante retirada deste quadro é que a grande maioria dos programas possuem pouco paralelismo intrínseco em seu código. Em termos de paralelismo nos limites dos blocos básicos essa conclusão já é de longa data conhecida. Entretanto, o que tenta-se mostrar aqui é que mesmo expandindo-se o escopo de busca além dos limites dos blocos básicos, através da análise dos blocos *controle-equivalentes*, o paralelismo resultante é pequeno. Assim, as formas mais adequadas de aumento de volume do paralelismo são aquelas em que código extra precisa ser gerado, tal como *loop unrolling*, *trace scheduling* e *software pipelining*.

Um resultado interessante aparece na coluna *BBs / Nó de região* para o programa *gcc*. Note que o valor obtido, *0.98*, mostra que existem mais nós de região do que blocos básicos. Isso significa que os mesmos blocos básicos estão ligados à diferentes nós de região. Esse fato ocorre em programas não estruturados e com muitas condições de controle. Por exemplo, o número de vezes em que o comando *goto* é utilizado no código-fonte do compilador *gcc* é razoavelmente grande.

Capítulo 4

Análise dos Pares Definições-Usos

4.1 Introdução

Neste capítulo, apresentamos estudos que mostram que grande parte das dependências de dados entre operações não apresentam distâncias longas, isto é, que o número de instruções entre a operação que define um registrador e a(s) operação(ões) que utiliza(m) este registrador é suficientemente pequeno para permitir uma efetiva utilização dos mecanismos de *bypassing* (possivelmente) adicionados à arquitetura.

Os resultados destes estudos dão sustentação ao algoritmo de escalonamento de instruções proposto no capítulo 5, que escalona instruções para uma arquitetura VLIW particionada com mecanismo de *bypassing*.

Este algoritmo tem como base permitir que linhas de *bypassing* de operandos entre diferentes unidades funcionais, sejam elas pertencentes a uma mesma partição ou não, sejam cautelosamente inseridas a fim de reduzirmos o número total de instruções de cópia entre partições.

A figura 4.1 mostra o grafo de dependências de dados de um fragmento de código do *benchmark jpeg*. Os nós do grafo representam instruções, e as arestas representam as dependências de dados verdadeiras entre as instruções[1]. Pode-se observar no exemplo que as dependências de dados presentes no código tendem a formar cadeias de arestas que ligam instruções próximas, e na maioria das vezes conectam uma operação fonte a poucas operações destino (na maioria das vezes somente uma). Instruções próximas são instruções que no código original (sem escalonamento, uma unidade funcional sendo considerada) seriam executadas numa janela de poucos ciclos de relógio.

A seguir são apresentados resultados da análise dos pares definição-usos das variáveis para os programas dos *benchmarks* utilizados no decorrer de todo esse trabalho. O código

[1] Do inglês, *true dependences*

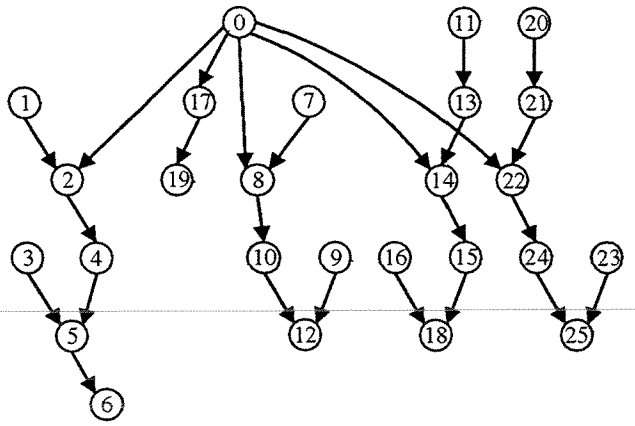


Figura 4.1: Fragmento de código extraído do *benchmark* jpeg.

utilizado nos estudos é gerado para uma máquina sequencial e tem como objetivo mostrar que as distâncias "definições-usos" das variáveis são pequenas.

O principal objetivo desta análise é medir o percentual de definições de registradores no código da aplicação que podem ser usados diretamente através da rede de *bypassing* de operandos.

Se esse percentual for razoavelmente grande, justifica-se o esforço em adicionar mecanismos de *bypassing* mesmo entre unidades funcionais pertencentes a diferentes *clusters* da arquitetura particionada. Isso possibilita que instruções com dependências de dados sejam escalonadas para serem executadas em diferentes *clusters* sem que seja necessário introduzir cópias de registradores entre os *clusters*.

4.2 Análises Realizadas

Foram realizadas as seguintes análises para os vários *benchmarks* adotados:

- Número médio de instruções entre a definição e o(s) uso(s) de um registrador.
- Número médio de usos de uma definição.
- Percentual das definições cujos usos encontram-se no mesmo bloco básico.
- Valores cumulativos mostrando o percentual de definições que tem usos distantes de até n instruções, n variando de 1 até 10.

Conforme ve-se nas figuras 4.2, 4.3 e 4.4, o percentual de definições consumidas dentro da janela das próximas 10 instruções é quase sempre maior do que 90%. Por este motivo as análises não ultrapassaram este valor.

Para realizar as análises citadas, uma implementação baseada na biblioteca de edição de executáveis *WARTS/EEL* [39] foi realizada. WARTS/EEL é uma biblioteca de classes C++ que tem por finalidade auxiliar na análise de programas já compilados para uma arquitetura RISC padrão. Embora as informações de *liveness analysis* pudessem ser realizadas pelo compilador, preferiu-se uma análise a partir do código-objeto gerado por este que já incorpora a alocação de registradores. Certamente esta metodologia é dependente do alocador de registradores e do algoritmo de escalonamento utilizado pelo compilador. Entretanto, consideramos a estratégia válida por se tratar de uma arquitetura-alvo RISC padrão, semelhante àquela que forma cada *datapath* da arquitetura VLIW sugerida no capítulo 2. Para cada programa dos *benchmarks* analisados:

1. Foi gerado o código-objeto a partir do código-fonte em C ou FORTRAN utilizando-se os compiladores GCC ou G77.

2. O código-objeto foi instrumentado utilizando-se a implementação baseada em EEL de forma a possibilitar a coleta dos dados desejados.

Em cada programa a instrumentação do código-objeto foi realizada construindo-se o grafo de fluxo de controle (CFG) e o grafo de dependência de dados (DFG) para cada função sendo compilada. O DFG é construído através da análise conhecida como *Definições Alcançáveis* (*Reaching Definitions*) [4]. Neste âmbito, WARTS/EEL somente provê bibliotecas para a construção do CFG. O suporte para a análise de *Reaching Definitions* teve de ser totalmente implementado. Este suporte consta de:

- Identificação dos registradores sendo escritos e lidos por cada instrução.
- Construção iterativa dos conjuntos *Gen* e *Kill* [4] de cada bloco básico.
- Construção iterativa dos conjuntos *In* e *Out* [4] de cada bloco básico.
- Alocação das arestas de *Use-Def* e *Def-Use* [4] entre instruções dependentes.

4.3 Resultados

A distribuição das distâncias entre a definição de um registrador e seu(s) o uso(s) nos *benchmarks* analisados é apresentada na tabela 4.1. Os gráficos correspondentes às distribuições cumulativas são mostrados nas figuras 4.2, 4.3 e 4.4 . Para cada benchmark analisado, as colunas da tabela 4.1 representam a percentagem de definições que são usadas nas L instruções seguintes, sendo L uma amplitude variando entre 1 e 10.

Benchmark	Programa	L=1 (%)	L=2 (%)	L=3 (%)	L=4 (%)	L=5 (%)	L=6 (%)	L=7 (%)	L=8 (%)	L=9 (%)	L=10 (%)
SPECInt 95	099.go	65.1	5.6	2.9	10.7	1.9	2.3	1.0	1.2	2.5	0.8
	124.m88ksim	64.5	7.1	8.3	3.5	2.8	2.5	2.0	1.3	1.1	0.9
	126.gcc	42.9	14.7	8.3	5.1	4.0	2.8	2.4	1.7	1.4	1.4
	129.compress	59.1	7.9	6.9	3.5	3.5	2.9	2.4	1.9	1.4	1.6
	130.li	71.3	11.4	7.6	3.1	1.6	1.5	0.8	0.5	0.5	0.5
	132.ijpeg	57.1	12.9	6.0	3.3	2.4	2.8	2.2	1.1	1.6	0.9
	134.perl	60.1	7.2	6.9	3.8	2.4	1.8	1.6	1.1	0.8	0.8
	147.vortex	57.6	11.4	8.9	6.5	2.8	2.2	2.7	1.0	0.9	0.9
SPECFp 95	101.tomcatv	36.0	16.4	9.7	7.3	6.7	5.6	3.0	2.0	1.9	1.7
	102.swim	37.6	15.7	9.4	7.0	6.5	5.3	3.2	1.9	1.8	1.6
	103.su2cor	47.3	12.7	7.9	5.3	5.5	4.4	2.5	1.7	1.6	1.4
	104.hydro2d	54.2	9.5	6.2	3.3	5.1	2.4	2.9	1.2	1.3	1.1
	107.mgrid	50.9	13.9	6.7	5.5	5.1	4.0	2.4	1.5	1.6	1.2
	110.applu	47.1	13.4	8.3	5.4	5.3	4.2	2.8	1.6	1.7	1.5
	125.turb3d	56.0	11.1	5.7	3.9	3.8	3.3	1.7	1.9	1.2	1.1
	141.apsi	65.4	10.2	3.1	3.3	2.8	2.1	1.7	1.4	1.0	1.0
	145.fpppp	47.2	18.3	9.8	4.4	4.0	3.3	1.9	1.3	1.2	1.1
	146.wave5	63.9	9.8	3.7	3.4	3.3	2.2	2.4	1.3	1.1	0.9
Misc.	gsm	32.4	16.8	9.8	6.5	6.3	4.4	3.5	2.0	1.8	1.3
	mpeg2dec	32.1	17.7	9.8	6.9	4.8	3.8	2.2	2.1	1.9	1.5
	mpeg2enc	29.6	15.4	10.4	6.4	5.2	3.8	2.6	2.3	2.2	1.9
	paraffins	70.1	10.7	3.6	6.5	2.8	0.9	1.4	0.3	0.7	0.1
	kalman	58.6	15.2	1.8	4.3	1.4	1.9	1.9	0.9	1.5	0.6
	fir	61.9	11.5	2.6	4.2	3.7	3.9	2.1	1.8	2.4	1.3
	dag	70.6	8.7	4.7	5.5	3.2	2.4	1.6	0.8	0.8	0.0
	eight	70.2	10.5	5.7	5.7	1.9	2.9	1.9	1.0	0.0	0.0

Tabela 4.1: Percentual das definições que são usadas dentro das L próximas instruções.

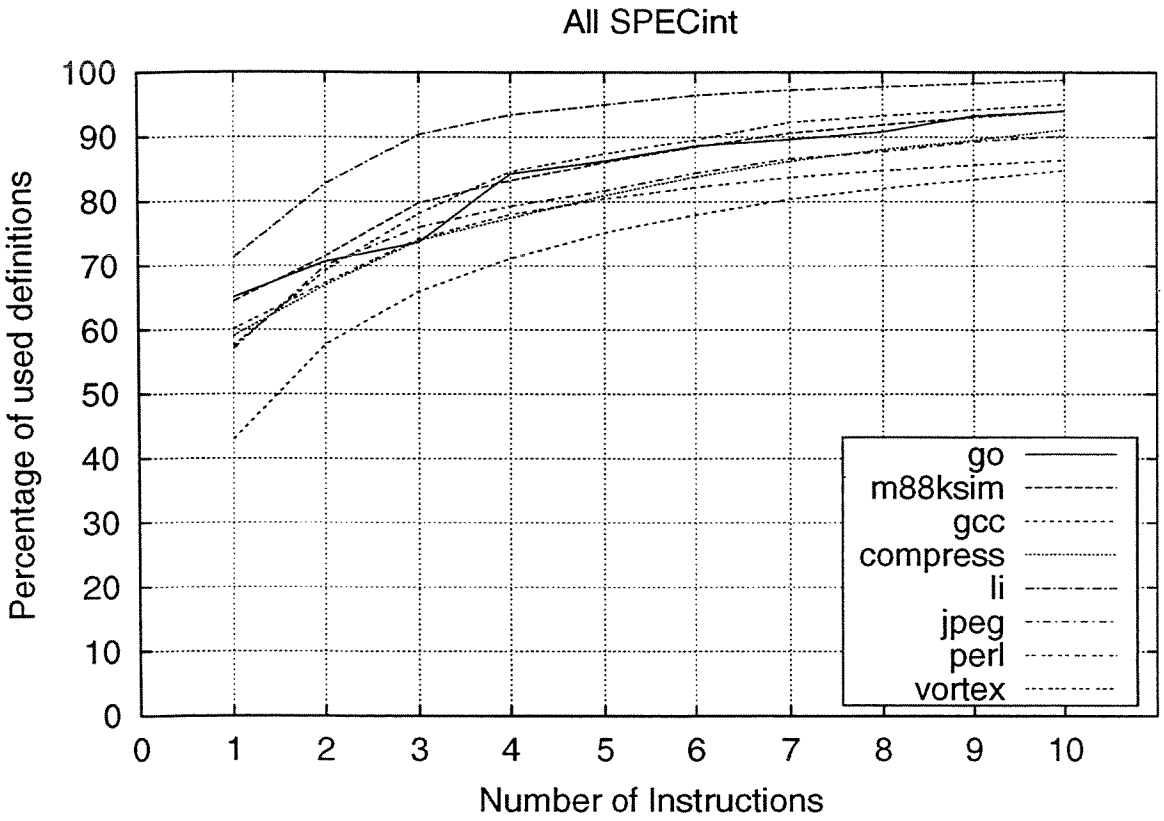


Figura 4.2: Distribuição cumulativa para o benchmark SPECINT95.

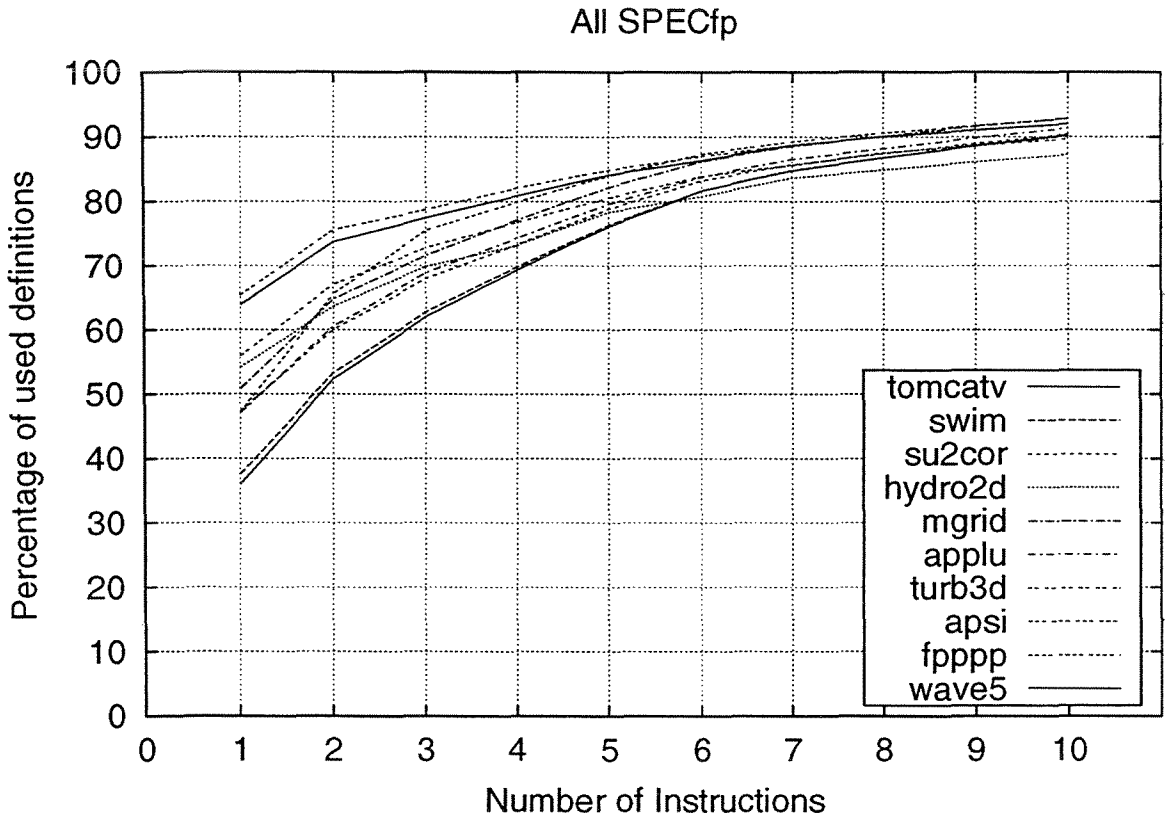


Figura 4.3: Distribuição cumulativa para o benchmark SPECFP95.

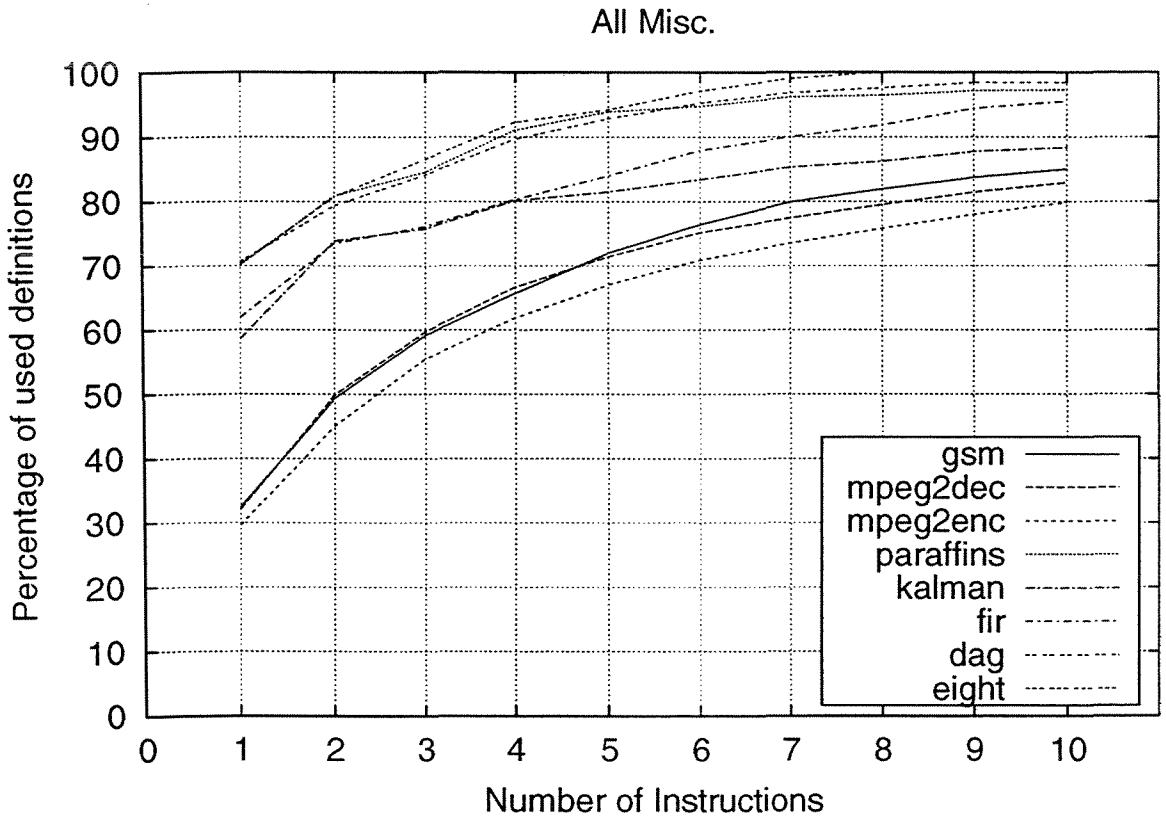


Figura 4.4: Distribuição cumulativa para o conjunto *Miscelaneous*.

Analisando-se os resultados apresentados neste capítulo podemos observar que, na grande maioria dos testes, um número expressivo das definições é usado nas 3 instruções seguintes. Na média, em 54.63% das instâncias, a distância desde a definição do registrador até seu uso é de apenas uma instrução. Ainda, se considerarmos distâncias de 2 e 3 instruções, esse valor sobe para 66.70% e 73.39%, respectivamente.

Com relação aos conjuntos distintos de *benchmarks*, podemos verificar que todos os programas do *benchmark* SPECInt possuem índices maiores do que 70% considerando-se distâncias de 3 instruções. Para o *benchmark* SPECfp, 60% das definições de todos os programas são usadas em uma janela das 3 instruções seguintes. Por último, para o conjunto *Miscelaneous*, 5 programas apresentam índices maiores do que 70% e 3 programas possuem índices maiores do que 55% para a mesma distância de 3 instruções sendo considerada.

Essa distância (3 instruções) é particularmente interessante pois a mesma representa o número de ciclos que um *pipeline* de 5 estágios mantém os resultados das operações nos registradores do *pipeline*. Entretanto, algumas instruções gastam mais do que um ciclo para serem executadas. Assim, a medida da distância em termos do número de *instruções* não representa uma métrica fiel à realidade. Todavia, Henessy [31] mostra que a grande maioria das instruções que são executadas nos programas são instruções de um ciclo[2] , indo ao encontro da metodologia RISC de projetar o microprocessador para as instruções mais executadas. Assim, a metodologia torna-se de certa forma válida e representa uma medida aproximada das distâncias reais entre definições e usos das variáveis.

Outro resultado interessante diz respeito ao número médio de usos de cada definição. Conforme mostrado na tabela 4.2, esse número se mantém consistentemente abaixo de 2 usos em todos os *benchmarks* analisados. Como última análise, computamos o percentual de usos que estão localizados no mesmo bloco básico de suas respectivas definições e o número médio de instruções entre a definição e o(s) uso(s) dos registradores.

[2]Ocupam cada estágio do *pipeline* durante um único ciclo

Benchmark	Programa	# médio de usos	% mesmo BB	# inst. entre def-use
SPECInt 95	099.go	1.10	81.37	2.77
	124.m88ksim	1.14	75.51	2.91
	126.gcc	1.76	36.41	3.80
	129.compress	1.24	69.73	2.51
	130.li	1.07	73.04	1.68
	132.ijpeg	1.25	73.19	3.03
	134.perl	1.31	60.29	4.17
	147.vortex	1.20	63.44	2.85
SPECFp 95	101.tomcatv	1.75	30.95	3.22
	102.swim	1.72	33.50	3.29
	103.su2cor	1.50	49.75	3.10
	104.hydro2d	1.39	66.79	3.67
	107.mgrid	1.46	50.89	3.10
	110.applu	1.54	48.35	3.38
	125.turb3d	1.39	60.04	3.31
	141.apsi	1.22	76.52	4.13
	145.fpppp	1.35	59.10	2.82
	146.wave5	1.23	74.5	3.42
Misc.	gsm	1.52	49.43	4.01
	mpeg2dec	1.62	42.38	3.95
	mpeg2enc	1.65	44.33	5.07
	paraffins	1.07	82.86	1.67
	kalman	1.23	78.46	2.75
	fir	1.27	72.18	1.84
	dag	1.23	60.32	1.43
	eight	1.25	58.65	1.33

Tabela 4.2: Número médio de usos de cada definição do programa, % dos usos cujas definições encontram-se no mesmo bloco básico e número médio de instruções entre definição-uso.

4.4 Efeitos das Técnicas de Aumento no Volume de Paralelismo

Considerando-se que as análises realizadas neste capítulo levam em conta programas originalmente seqüenciais e compilados para uma arquitetura escalar, um aumento no volume de paralelismo tendo em vista uma arquitetura paralela poderia comprometer os resultados apresentados. Imagine, por exemplo, uma arquitetura particionada com 4 unidades funcionais. Suponha que um laço sendo escalonado foi desenrolado 8 vezes através da aplicação de *loop unrolling*. Considere ainda que o algoritmo de escalonamento escolheu alocar, em um mesmo ciclo, nas 4 unidades funcionais disponíveis, a mesma primeira operação *op1* das 4 primeiras replicações do laço. No ciclo seguinte, o algoritmo escolhe alocar novamente as operações *op1* das 4 replicações restantes do laço. Desta forma, nos dois ciclos considerados teremos várias versões da operação *op1* sendo executadas.

Por último, imagine uma operação *op2* que depende do resultado da operação *op1* presente no laço original. Aquelas operações *op1* das primeiras 4 replicações do laço foram executadas no primeiro ciclo, e as respectivas operaçôes *op2* somente terão chance de ser executadas dois ciclos a frente, no melhor caso. No escalonamento original, seqüencial, esta operação poderia entrar no *pipeline* no ciclo seguinte à execução de *op1*. Entretanto, após a aplicação de *loop unrolling* todas as unidades funcionais foram ocupadas durante dois ciclos por várias versões de uma mesma operação, fazendo com que a distância entre *op1* e *op2* de cada replicação aumentasse.

Se o aumento de paralelismo não for realizado de uma forma inteligente, isso pode significar em uma não utilização do mecanismo de *bypassing* devido a um acréscimo na distância entre operações dependentes. Entretanto, se esse aumento de paralelismo for realizado tentando-se balancear o paralelismo exposto com o paralelismo disponível na arquitetura, isso não acontece. Uma das formas de balancear o volume de paralelismo disponível em uma máquina VLIW particionada e o exposto pela aplicação é desenrolar os laços um número de vezes igual ao número de *clusters* da arquitetura. Desta maneira, as várias iterações do laço poderão ser naturalmente distribuídas pelos *clusters* da arquitetura [58, 57, 59]. Isso fará com que o comportamento das dependências de dados analisadas neste capítulo seja "replicado" nos *clusters*, mantendo a distância média entre definições e usos das variáveis dos programas.

Capítulo 5

Escalonamento de Instruções em Arquiteturas VLIW Particionadas

5.1 Escalonamento de Instruções

De uma maneira geral, o escalonamento de instruções para arquiteturas particionadas é uma tarefa mais complexa do que o escalonamento de instruções para processadores não-particionados. Em máquinas com apenas um banco de registradores a tarefa de escalonamento resume-se em, repetidamente, encontrar a operação pronta com maior prioridade e alocá-la a uma unidade funcional disponível, atualizando sempre as estruturas de dados que refletem o estado do processador.

Em arquiteturas particionadas, além da maior dificuldade na tarefa de escalonamento de instruções, a comunicação limitada entre partições tem uma influência significativa no resultado final do escalonamento. Instruções de cópias entre diferentes *clusters* da arquitetura particionada poderão ser eventualmente necessárias para resolver certas dependências de dados. Essas cópias provavelmente aumentarão o número total de ciclos para executar uma aplicação.

Várias técnicas locais e globais endereçam o problema de escalonamento de instruções para máquinas VLIW *não-particionadas*. Como exemplos podemos citar *Trace Scheduling* [22], *List Scheduling* [12], *Critical Path Scheduling* [15] e *Percolation Scheduling* [45]. Entretanto, como estes algoritmos não foram projetados para arquiteturas particionadas, uma fase posterior de particionamento é necessária.

Neste capítulo apresentamos uma técnica local de escalonamento de instruções para uma arquitetura VLIW particionada, formada por duas fases que permite, na última delas, "inserir" linhas de *bypassing* de operandos entre as unidades funcionais mais comunicantes. Por unidades funcionais comunicantes chamamos os pares de unidades funcionais onde operações dependentes são alocadas.

5.1.1 Definição do Problema

O problema de escalonamento de instruções endereçado neste capítulo pode ser enunciado da seguinte maneira:

Seja B um (macro) bloco básico, representado por um grafo de fluxo de dados ponderado $G = (V, E, w)$. Cada aresta $(u, v) \in E$ é ponderada por um inteiro $w(u,v)$ que representa um valor de latência de se executar a operação u e transferir o resultado como operando para a operação v. Assume-se que o grafo é gerado após a fase de seleção de instruções, ou seja, os nós do DFG representam instruções concretas. As arestas representam dependências de dados [4] existentes entre as instruções.

Desta forma, a solução do problema de escalonamento é associar, para cada cada operação $v \in V$, uma unidade funcional específica e um ciclo de relógio determinado no qual a sua execução terá início.

Assume-se ainda que os nós (instruções) não foram associados as partições. Desta forma, um particionamento $P \rightarrow 1, .., Max$ entre os *Max clusters* deve ainda ser computado.

Como a arquitetura-alvo sendo considerada pressupõe a existência de comunicação (limitada) entre as partições, sempre que uma dependência de dados entre duas instruções ultrapassar a fronteira de duas partições diferentes, uma instrução de cópia deve ser gerada. Todavia, se existir a possibilidade de comunicação direta entre os dois *clusters*, via rede de *bypassing* de operandos, esta deve ser priorizada em relação a cópia.

Um *macro* bloco básico como tratado aqui é definido como sendo a união de diferentes blocos básicos através da utilização de técnicas de compilação bem definidas. Algumas das técnicas mais conhecidas são formação de superblocos e formação de hiperblocos. Para a implementação do algoritmo de escalonamento de instruções exposta neste capítulo foi utilizado o compilador IMPACT [32], proveniente da University of Illinois at Urbana-Champaign. Este compilador implementa mecanismos de *Loop unrolling* aliado à formação de super-blocos ou hiper-blocos. Através de passagem de parâmetros pode-se escolher entre uma combinação ou outra. A combinação escolhida para a implementação foi *loop-unrolling* mais formação de super-blocos. A formação de hiper-blocos não foi considerada pois neste caso a arquitetura-alvo deveria proporcionar a característica de predicação [44]. Embora isso não aumente a complexidade do problema, preferiu-se adotar uma arquitetura mais simples.

5.2 Algoritmo Proposto

A figura 5.1 mostra o algoritmo proposto para o escalonamento de instruções de um macro bloco básico. Este algoritmo é uma versão modificada do algoritmo de *List Scheduling* [12]. A arquitetura-alvo sendo considerada é uma máquina VLIW particionada, homogênea,

formada por F unidades funcionais e R bancos de registradores (*clusters*). Entretanto, os subconjuntos distintos de unidades funcionais que compõem os *clusters* da arquitetura não são pré-estabelecidos. Ao contrário, um dos *resultados finais* do algoritmo de escalonamento é informar quais unidades funcionais deverão ser reunidas entre R *clusters* da arquitetura, e entre quais unidades haverá linhas de *bypassing* de operandos.

Outra característica do algoritmo proposto é o de desacoplar a distribuição das unidades entre os *clusters* e a alocação da rede de *bypassing* de operandos. A alocação das linhas de *bypassing* será realizada entre as unidades funcionais mais comunicantes, sejam elas pertencentes a um mesmo *cluster* ou não. Geralmente, essas unidades serão justamente aquelas agrupadas em *clusters*. Porém, podem acontecer casos onde unidades de um mesmo *cluster* não possuem *bypass* e unidades de *clusters* diferentes sim, em prol de uma melhor decisão global. Isso é revisto na seção 5.4.

Os parâmetros a serem passados ao algoritmo de escalonamento são F, número de unidades funcionais, R, número de bancos de registradores (ou *clusters*) da arquitetura e B, número máximo de linhas de *bypassing* de operandos. Uma linha de *bypassing* de operandos é definida como sendo a ligação bilateral entre duas unidades funcionais conforme mostrado na figura 2.10.b.

```
(1)  for each macro_block in set of function's macro_blocks do
(2)
(3)      /* Calculate the priorities for this macro_block's operations */
(4)      calculate_priorities (macro_block);
(5)
(6)      for (current_cycle = 0 ;
(7)           macro_block->number_of_unscheduled_operations > 0;
(8)             current_cycle++)
(9)      {
(10)
(11)         while (1)
(12)         {
(13)             /* Test if there is al least one available */
(14)             /* functional unit in this cycle */
(15)             if (number_of_occupied_slots (current_cycle) == number_of_fus)
(16)                break;
(17)
(18)             /* Find the highest priority ready operation   */
(19)             sched_op = find_highest_priority_ready_operation
(20)                         (macro_block, current_cycle);
(21)
(22)             /* If there is no ready operation */
(23)             /* in this cycle goto next cycle */
(24)             if (sched_op == NULL)
(25)                break;
(26)
(27)             /* Try to schedule sched_op in the first available */
(28)             /* functional unit that minimizes the execution latency. */
(29)             operation_scheduled = try_to_schedule_operation
(30)                                   (sched_op, current_cycle);
(31)
(32)             if (operation_scheduled == 1)
(33)             {
(34)                add_intercommunication
(35)                (sched_op, macro_block->intercommunication_table);
(36)             }
(37)         }
(38)     }
```

```
(38)
(39)        if (number_of_fus > 1)
(40)        find_most_communicating_fus
(41)              (macro_block, macro_block->intercommunication_table);
(42)
(43)  done;
```

Figura 5.1: Escalonamento de instruções para uma arquitetura VLIW particionada.

Na linha (1) o algoritmo inicia iterando sobre todos os macro-blocos da função sendo compilada. A linha (4) invoca a função para calcular as prioridades das operações do macro bloco sendo escalonado. A ordem de prioridades estabelecida é a seguinte:

1. Caminho crítico.
2. Número de operações dependentes.
3. Seqüência de escrita no código original.

A partir daí, na linha (6), outro laço do algoritmo é executado até que todas as operações do macro-bloco corrente tenham sido escalonadas. Neste laço, após testar se existe alguma unidade funcional disponível no ciclo corrente (linha 15), toma-se a operação-pronta de maior prioridade. [1]

Pode acontecer que no ciclo corrente não exista nenhuma operação-pronta devido a eventuais dependências de dados com operações-fonte que gastam vários ciclos para serem executadas. Neste caso, passa-se para o próximo ciclo. Se, ao contrário, existir pelo menos uma operação pronta no ciclo corrente, tenta-se associar esta operação a uma unidade funcional disponível e que minimize o número de ciclos de execução. As unidades funcionais que minimizam o número de ciclos de execução são aquelas onde foram escalonadas as operações-fonte das dependências de dados, ou seja, que definem os operandos da operação sendo escalonada.

Por exemplo, suponha o grafo representado na figura 5.2. Neste grafo, os nós representam operações e as arestas representam dependências de dados entre as operações. Suponha ainda que a operação A tenha sido escalonada para ser executada na unidade funcional denominada FU 1 e a operação B escalonada na unidade funcional denominada FU 2. Desta forma, as unidades funcionais preferenciais para a execução da operação C, que minimizam o número de ciclos, são FU 1 ou FU 2.

[1] Por operação-pronta entende-se uma operação em que todas as dependências de dados tenham sido resolvidas e que o ciclo corrente esteja dentro da faixa ALAP-ASAP da operação.

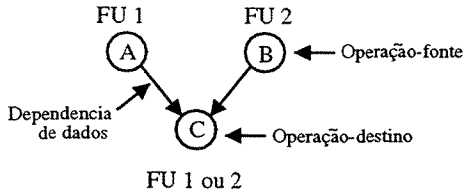


Figura 5.2: Grafo de dependência de dados entre operações a serem escalonadas.

Esta é uma das tarefas da função *try_to_schedule_operation*, ou seja, tentar alocar as operações dependentes nas mesmas unidades funcionais. Desta forma, uma das unidades FU 1 ou FU 2 deve ser escolhida para a operação C para que o algoritmo prossiga. Caso isso não seja possível, isto é, caso as unidades preferenciais da operação C estejam ocupadas, procura-se a próxima unidade funcional disponível a partir de uma lista de unidades funcionais. A informação da ocupação das unidades funcionais em cada ciclo é extraída da *reservation_table* que é associada a cada macro-bloco básico. Essa tabela mantém a informação das operações já escalonadas e reserva as unidades funcionais pelo número de ciclos necessário para a execução de cada operação. Um exemplo da mesma é mostrada na figura 5.6.

Nas linhas (34-35) do algoritmo, a função *add_intercommunication* é invocada caso a operação *sched_op* tenha sido efetivamente escalonada, isto é, caso o algoritmo tenha encontrado pelo menos uma unidade funcional disponível no ciclo atual. Esta subrotina tem a finalidade de percorrer todas as dependências de dados de entrada de *sched_op* e, para cada uma, verificar a unidade funcional da operação-fonte de dependência. Caso esta seja diferente da unidade funcional atribuída para *sched_op*, pode-se notar que houve uma comunicação entre duas unidades funcionais diferentes: duas operações dependentes foram escalonadas em duas unidades funcionais distintas. Assim, a tabela denominada *intercommunication_table* é atualizada refletindo essa comunicação. Essa é uma tabela que cada macro bloco possui para representar, sob a forma de uma matriz, a quantidade de intercomunicações que ocorreram entre os diferentes pares de unidades funcionais durante o escalonamento.

A figura 5.3 mostra uma seqüência de operações sendo escalonadas salientando a atualização da tabela *intercommunication_table*. Suponha que as decisões de escalonamento foram tais que as unidades funcionais escolhidas para as operações C, E, G e I são as mostradas na figura. As operações A, B, D, F e H são operações independentes, ou seja, não apresentam dependências de dados de entrada. Desta forma, a associação das unidades funcionais preferenciais das mesmas é realizada numa fase de inicialização do algoritmo.

Conforme o escalonamento vai sendo realizado, unidades funcionais vão sendo associadas às operações, sempre tentando-se associar para uma dada operação uma unidade

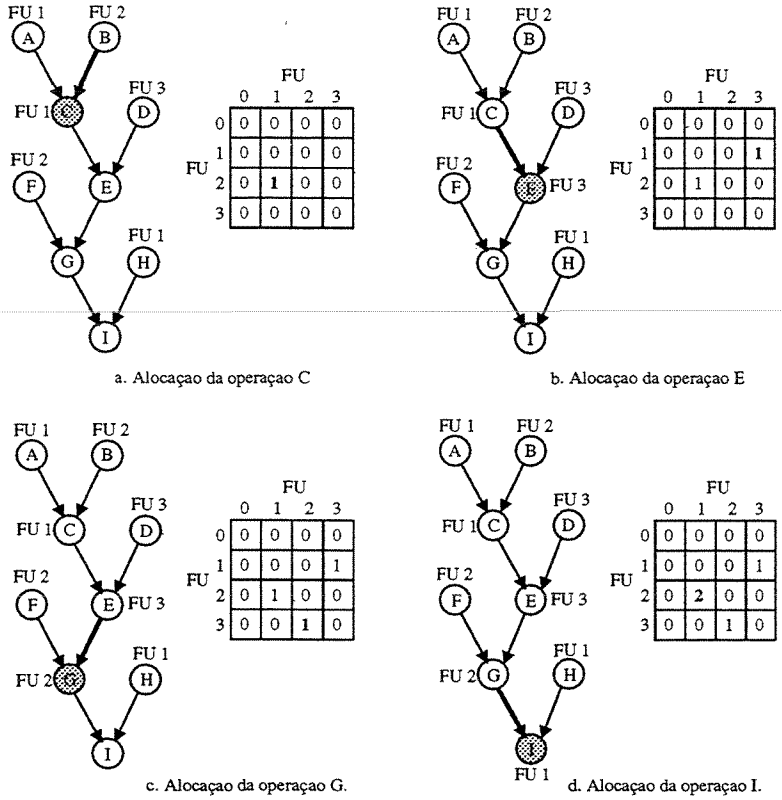


Figura 5.3: Mecanismo de atualização da tabela de intercomunicações.

funcional de sua preferência. Simultaneamente, a tabela de intercomunicações vai sendo atualizada a fim de refletir a quantidade de comunicações entre os diferentes pares de unidades funcionais. A tabela *reservation_table* também é constantemente atualizada no sentido de sinalizar quais unidades permanecerão ocupadas e por quantos ciclos de relógio isso ocorrerá.

Essa etapa termina após todas as operações do macro bloco terem sido escalonadas. Neste ponto, todas as operações têm suas unidades funcionais atribuídas, a *resevatiom_table* tem a "fotografia" do escalonamento e a tabela de intercomunicação reflete o volume de comunicação entre os diferentes pares de unidades funcionais.

A linha (40) do algoritmo desencadeia os próximos passos a serem executados. Nesta etapa são determinados os pares de unidades funcionais mais comunicantes, ou seja, aqueles pares de unidades com uma maior quantidade de operações dependentes escalonadas tal que a operação-fonte tenha sido escalonada em uma unidade funcional e a operação-destino na outra unidade funcional. O resultado final desta etapa será a distribuição das unidades funcionais entre os *clusters* da arquitetura sob o ponto de vista do macro-bloco recém escalonado. Em outras palavras, "como" o bloco recém escalonado preferiria que

a distribuição fosse realizada.

As informações computados pela função *find_most_communicating_fus* são explicadas e comentadas passo-a-passo a seguir, considerando-se como exemplo uma arquitetura com 4 unidades funcionais e 2 bancos de registradores (portanto, 2 *clusters*):

1 : Zerar a diagonal principal da matriz *intercommunication_table*, pois comunicações de uma unidade para ela mesma não são relevantes.

2 : Transformar a matriz *intercommunication_table* em uma matriz diagonal inferior, somando-se as células correspondentes aos mesmos pares de unidades funcionais. Isto é, a célula que tem o valor da intercomunicação entre as unidades i → j é somada com a célula j → i. O resultado é armazenado na célula [j][i] e a célula [i][j] é zerada. Este passo é mostrado na figura 5.4.

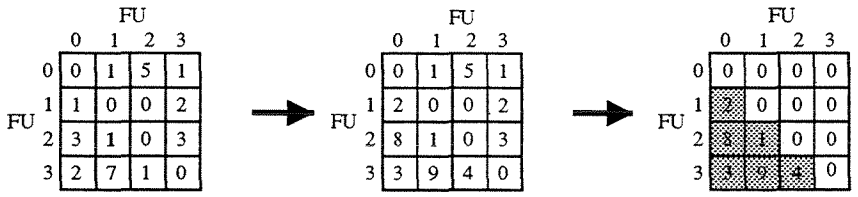
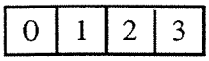


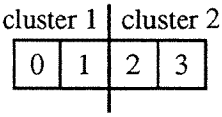
Figura 5.4: Passo 2 da função *find_most_communicating_fus*.

3 : Computar quais unidades funcionais formarão os *clusters* de forma a minimizar o número de comunicações entre diferentes *clusters*. Esta fase é ilustrada na figura 5.5. Primeiramente, um vetor com um número de células igual ao número de unidades funcionais da arquitetura é inicializado (cada célula contém um número que identifica cada uma das unidades funcionais da arquitetura - figura 5.5.a). Este vetor, após finalizada a etapa 3, representará a distribuição das unidades funcionais entre os *clusters*. Isto é, se o vetor contiver 4 posições (portanto 4 unidades funcionais) e a arquitetura sendo considerada possuir 2 *clusters*, as duas primeiras posições do vetor conterão as unidades funcionais do primeiro *cluster* e as duas posições seguintes representarão o segundo *cluster*. A figura 5.5.b mostra o vetor inicial dividido ao meio a fim de ilustrar a arquitetura considerada neste exemplo.

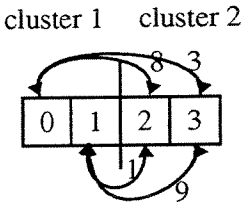
A distribuição final das unidades funcionais entre os *clusters* da arquitetura é computada utilizando-se um algoritmo de particionamento semelhante a LPK [40]. Neste algoritmo, repetidas trocas de unidades funcionais entre os *clusters* vão sendo realizadas e, a cada troca, um valor de custo é calculado. O valor de custo é computado como sendo o número total de intercomunicações que transpassam a fronteira entre as partições. O objetivo final das trocas é atingir uma distribuição que minimize o custo. A figura 5.5.c mostra como o custo total do vetor da figura 5.5.b é calculado utilizando-se a matriz *intercommunication_table*. Passos intermediários do algoritmo são mostrados nas figuras 5.5.d, 5.5.e, 5.5.f e 5.5.g. O resultado final para este exemplo é mostrado na figura 5.5.h.

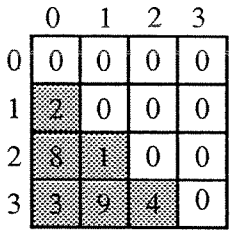


a. Inicializaçao do vetor

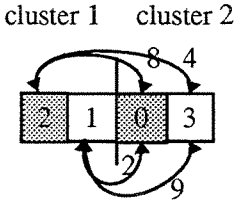


b. Particionamento inicial definido

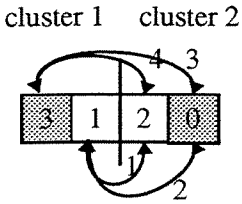




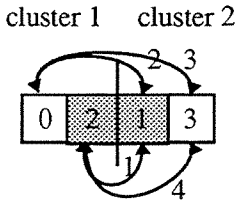
c. Custo da distribuiçao inicial = 21



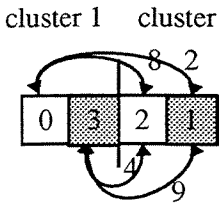
d. Primeira troca. Custo = 23.



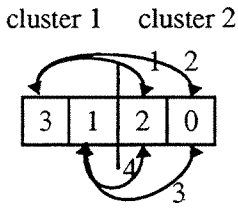
e. Segunda troca. Custo = 10.



f. Terceira troca. Custo = 10.



g. Quarta troca. Custo = 23.



h. Custo final = 10

Figura 5.5: Passo 3 da função *find_most_communicating_fus*.

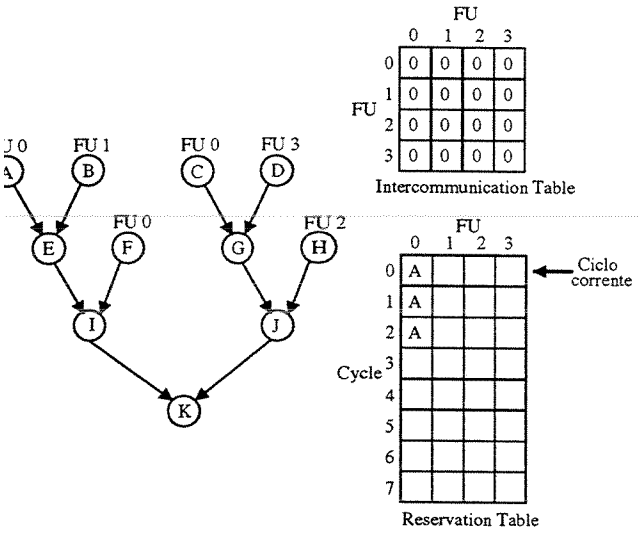
5.3 Um Exemplo de Escalonamento

Um exemplo com detalhes de um macro-bloco sendo escalonado é mostrado na figura 5.6. O macro bloco é representado sob a forma de um grafo onde as operações são os nós e as dependências de dados são as arestas. As unidades funcionais preferenciais das operações independentes são inicializadas (seqüencialmente) e o algoritmo da figura 5.1 entra em execução. Suponha que, após o cálculo de prioridades, a ordem das operações segue a ordem alfabética referente aos rótulos das mesmas, conforme mostrado na figura 5.6. Assim, a operação "A" é a primeira da lista de prioridades. Como a unidade preferencial desta operação, FU0, encontra-se disponível no ciclo 0, e a operação "A" faz parte da lista de operações-prontas, esta é escalonada para ser executada na sua unidade preferencial. A *reservation_table* é atualizada levando-se em conta o número de ciclos necessários para a execução de "A" (3). A figura 5.6.a mostra o estado atual das tabelas *reservation_table* e *intercommunication_table* neste ponto. Como a operação "A" não apresenta dependências de dados de entrada, o próximo passo é escalonar a operação "B". Esta operação tem como unidade funcional preferencial a unidade FU1. Como "B" também está pronta no ciclo 0, e sua unidade preferencial está disponível a partir deste ciclo, "B" é escalonada para ser executada em FU1. A figura 5.6.b ilustra este ponto do escalonamento (suponha que "B" gasta 2 ciclos).

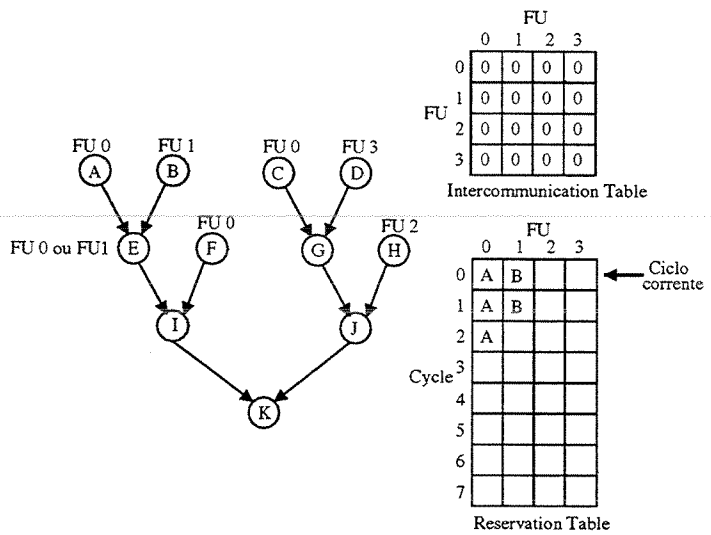
A próxima operação-pronta a ser considerada é a operação "C". Note que a unidade funcional preferida por esta unidade, FU0, está ocupada no ciclo de relógio atual. Desta forma, o algoritmo busca a primeira unidade funcional disponível para escalonar "C", ou seja, FU2. Para a operação "D", nada de novo acontece. A figura 5.6.c mostra o ponto em que "D" é escalonado. Note que as unidades funcionais preferenciais para a operação G são atribuídas neste instante, ou seja, após ter-se a informação das unidades funcionais atribuídas a "C" e "D".

O mesmo raciocínio é utilizado para as demais operações, e o resultado final é mostrado na figura 5.6.d. Note que intercomunicações são adicionadas a tabela *intercommunication_table* sempre que duas operações dependentes são escalonadas em diferentes unidades funcionais. Na figura 5.6.d pode-se observar as dependências de dados entre operações dependentes que foram escalonadas para serem executadas em unidades funcionais distintas.

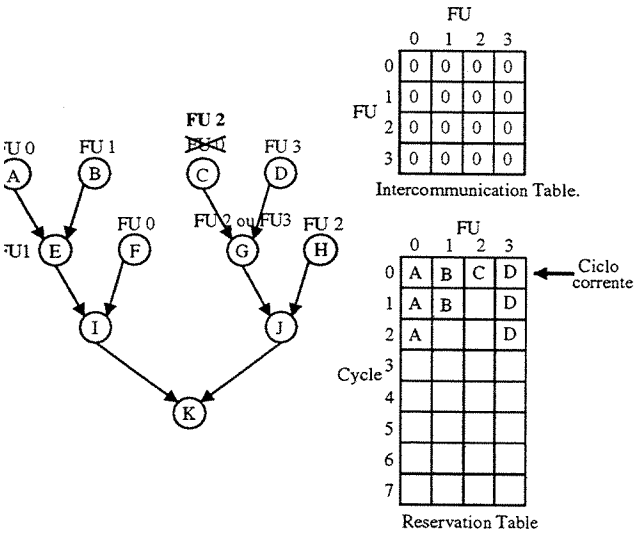
Note que nenhuma instrução de cópia foi emitida para transferir operandos entre operações. Isso ocorre porque durante o decorrer do escalonamento as unidades funcionais que formam cada *cluster* ainda não estão definidas. Conforme citado, esta é uma das respostas do algoritmo de escalonamento sendo proposto. As instruções de cópia necessárias serão inseridas numa fase pós-escalonamento, onde a configuração da arquitetura já é conhecida.



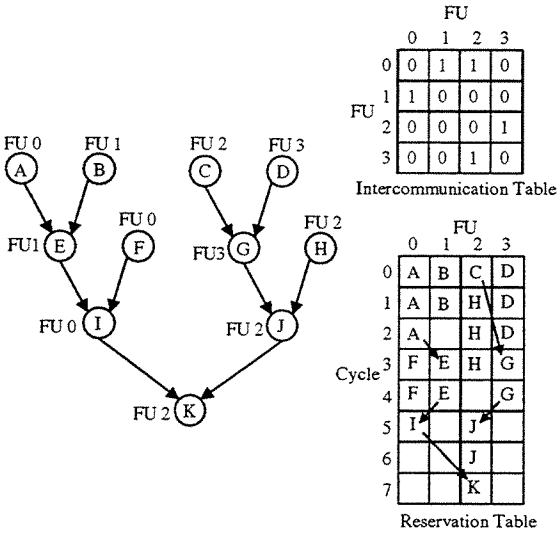
a.Estado após o escalonamento da operação "A".



b.Estado após o escalonamento da operação "B".



c.Estado após o escalonamento das operações "C" e "D".



d.Estado após o escalonamento de todas operações.

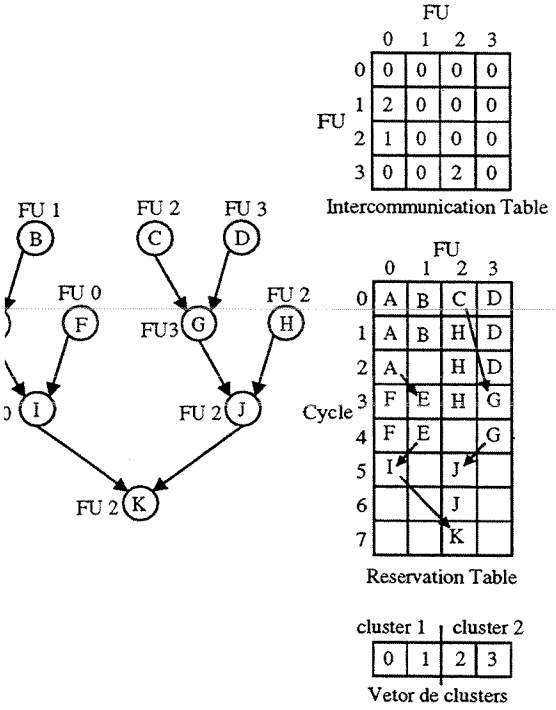
Figura 5.6: Escalonamento passo-a-passo de um macro bloco.

5.4 Alocação da rede de *bypassing*

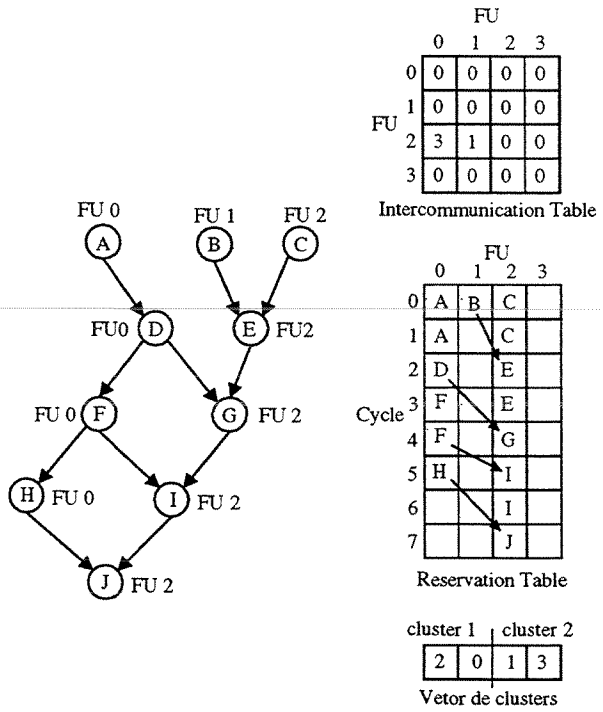
Conforme citado anteriormente, linhas de *bypassing* são adicionadas entre os pares de unidades funcionais mais comunicantes com a finalidade de proporcionar uma transparência de operandos entre as mesmas. Entretanto, essa *não* é uma decisão local, pertencente a cada macro bloco. Conforme visto até o momento, existe uma *intercommunication_table* e um vetor representando os *clusters* para cada macro bloco básico, de cada função, do programa sendo compilado. Entretanto, como as linhas de *bypassing* são físicas, e fixas, uma decisão global, baseada nas *intercommunication_tables* de todos os macro bloco básicos, deve ser tomada.

A figura 5.7 mostra as *reservation_table*, as *intercommunication_table* e os vetores de *clusters* para alguns macro blocos de um programa sendo compilado após terem sido escalonados. Note na figura que os *clusters* para cada macro bloco foram organizados de forma diferente, correspondendo às comunicações daquele macro bloco.

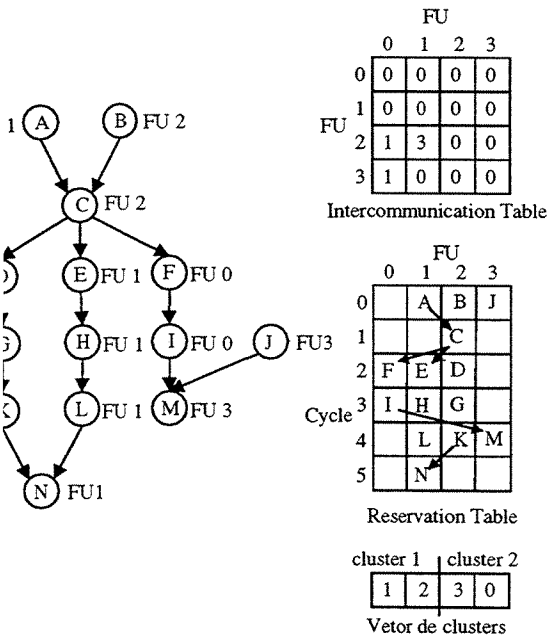
Entretanto, os rótulos dados às unidades funcionais em cada macro bloco não são muito conclusivos, pois são de uma certa forma "aleatórios" quando comparamos dois blocos distintos. Em outras palavras, os "rótulos" das unidades funcionais atrbuídas às operações independentes no início do algoritmo de escalonamento são de certa forma propagados para as outras operações no grafo.



a.Macro bloco básico 1.



b.Macro bloco básico 2.



c.Macro bloco básico 3.

Figura 5.7: Vários macro blocos básicos do programa.

Desta maneira, consideramos que a organização dos *clusters* em cada macro bloco é na realidade apenas uma organização "virtual" das unidades funcionais. Fisicamente, no *hardware*, existem quatro unidades funcionais, as quais denominaremos $dp1, $dp2, $dp3 e $dp4, que estão em posições fixas e imutáveis. O vetor que representa os *clusters* de cada macro bloco indica apenas um mapeamento entre as unidades funcionais "virtuais" daquele bloco com relação às unidades funcionais implementadas no circuito integrado. A figura 5.8 mostra um vetor de *clusters* das unidades funcionais "físicas" sendo considerado.

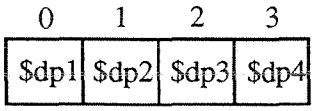


Figura 5.8: Vetor de *clusters* relativo às unidades fisicamente implementadas .

Desta forma, cada vetor de *clusters* de cada macro bloco básico mostrado na figura 5.7 deve ser mapeado para o vetor de *clusters* no *hardware* mostrado na figura 5.8. Assim, para o macro bloco básico 1 da figura 5.7 a unidade funcional 0 na realidade representa a unidade funcional *física* $dp1. A mesma unidade funcional *física* $dp1 para o macro bloco básico 2 é representada naquele bloco pela unidade funcional "virtual" 2. O mesmo raciocínio vale para realizar o mapeamento de todas as unidades funcionais virtuais para físicas.

Assim sendo, a rede de *bypassing* de operandos deve ser alocada considerando-se as unidades funcionais *físicas*, e não virtuais, de cada macro bloco. Em outras palavras, a quantidade de intercomuicação representada pela *intercommunication_table* de cada macro bloco deve ser convertida para uma *hardwired_intercommunication_table*, ou seja, uma única tabela de comunicações relativa às unidades físicas.

Observe portanto a *intercommunication_table* da figura 5.7.b. Nesta tabela, ve-se uma comunicação de operandos entre as unidades funcionais "virtuais" 2 e 0. Com o mapeamento, vemos que na realidade esta comunicação de operandos ocorreu entre as unidades *físicas* $dp1 e $dp2.

Desta forma, para computar-se a tabela *hardwired_intercommunication_table* somam-se todas as *intercommunication_tables* de todos os macro bloco básicos, realizando os mapeamentos entre unidades virtuais e físicas necessários.

A figura 5.9 mostra o cálculo da *hardwired_intercommunication_table* para o exemplo da figura 5.7.

Outro fator que deve ser levado em conta no cálculo da *hardwired_intercommunication_table* é o nível do aninhamento de laços em que se encontra cada macro bloco. Isto é, a

intercommunication_table de cada macro bloco reflete os valores absolutos de comunicação daquela porção de código. Entretanto, quando comparam-se dois macro blocos diferentes, pode ser que os mesmos façam parte de níveis diferentes de dois laços do programa. Assim, o macro bloco pertencente ao laço mais interno será executado mais vezes que o macro bloco do laço mais externo. Para solucionar este impasse, o nível de aninhamento do laço[2] que contém o macro bloco é levado em consideração *(Nesting Level)*.

[2]Na realidade, dois laços podem conter o mesmo macro bloco básico caso os mesmos sejam aninhados entre si. Neste caso, o laço *mais interno* que contém o macro bloco é considerado.

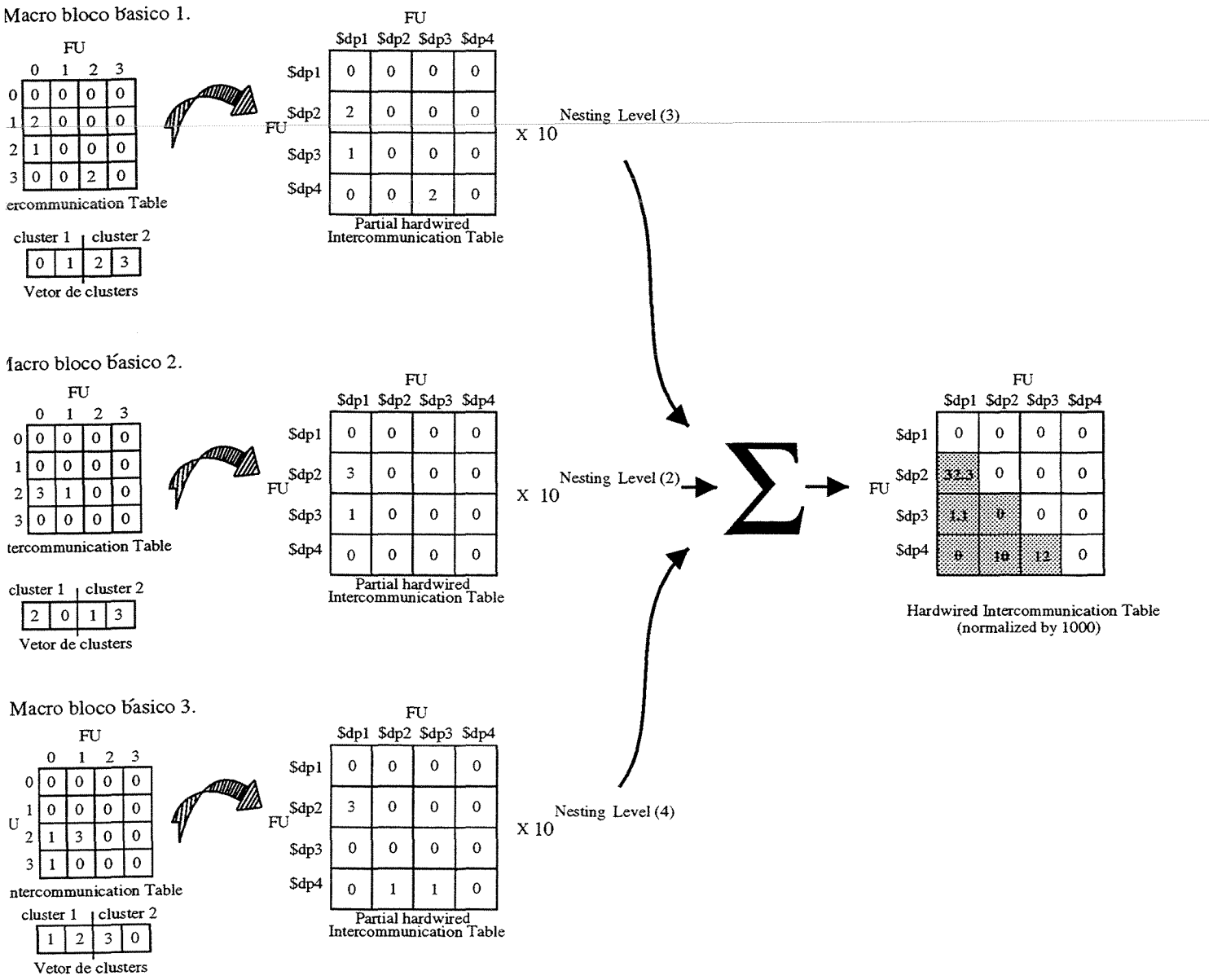


Figura 5.9: Mapeamento entre unidades funcionais "virtuais" e "físicas" .

Com a *hardwired_intercommunication_table* em mãos, a determinação de quais pares de unidades funcionais físicas terão *bypassing* é simples. Os "$MAX\_BYPASSING$" pares de unidades funcionais mais comunicantes terão linhas de *bypassing* de operandos adicionadas no *hardware*. $MAX\_BYPASSING$ é um valor passado como parâmetro para o algoritmo de escalonamento de instruções. Nos resultados experimentais apresentados no final deste capítulo, foram considerados vários valores para este parâmetro com a finalidade de medir o comportamento da aplicação conforme a arquitetura varia o seu nível de intercomunicação.

Assim, basta ordenar as células hachuradas na *hardwired_intercommunication_table* mostrada na figura 5.9 para determinar quais os pares de unidades funcionais físicas terão *bypassing* adicionado. O algoritmo utilizado para o ordenamento é o *bublesort*, devido ao número reduzido de pares de unidades funcionais possível mesmo considerando uma arquitetura com várias unidades funcionais. Este passo é ilustrado na figura 5.10.

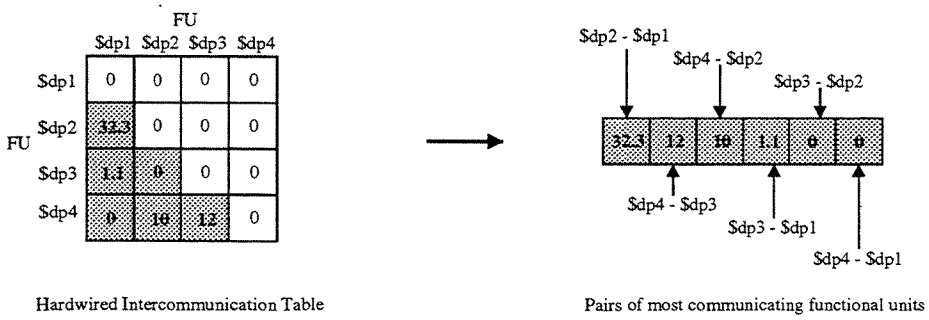


Figura 5.10: Determinação dos pares de unidades funcionais que terão *bypassing* .

A partir da figura 5.10 pode-se extrair os pares de unidades funcionais físicas mais comunicantes. Desta forma, linhas de *bypassing* seguindo-se o modelo da figura 2.10 são adicionadas entre os $MAX\_BYPASSING$ pares.

Note que a *hardwired_intercommunication_table* foi utilizada de uma forma diferente das *intercommunication_tables* dos macro bloco básicos isolados. Naqueles, a *intercommunication_table* era usada como fonte de valores para a distribuição das unidades funcionais entre os *clusters* da arquitetura. Agora, a tabela é utilizada para a decisão de quais pares de unidades funcionais físicas terão *bypassing* de operandos. Os dois problemas, apesar de intrinsicamente relacionados, são tratados de forma independente.

Note que é possível que duas unidades funcionais físicas pertencentes a diferentes *clusters* apresentem *bypassing* e que duas unidades pertencentes ao mesmo *cluster* não possuam esta facilidade. Isso é devido ao algoritmo LPK de particionamento aplicado nas unidades virtuais priorizar uma solução "global" com um menor custo.

Vale relembrar que a rede de *bypassing* "dentro" de um único *pipeline* é sempre considerada padrão. Em outras palavras, o que está sendo especializado aqui são as linhas de *bypassing entre* unidades funcionais diferentes. Entretanto, num mesmo *pipeline* considera-se sempre a existência de *bypassing*. Caso contrário, mesmo que as operações fossem escalonadas na mesma unidade funcional teríamos *data hazards* freqüentemente.

5.5 Mapeamento Reverso e Inserção das Instruções de Cópias

Até este ponto o algoritmo de escalonamento de instruções apresentado não considerou a inserção de cópias porque não tinha conhecimento de quais unidades funcionais faziam parte de quais *clusters*. Uma terceira etapa do algoritmo trata da questão de mapeamento reverso e inserção de cópias.

5.5.1 Mapeamento Reverso

Da figura 5.10 pode-se extrair os pares de unidades funcionais *físicas* mais comunicantes que terão linhas de *bypassing* para transparência de operandos. Entretanto, utilizando um raciocínio similar ao mapeamento mostrado na figura 5.9, as unidades funcionais *virtuais* de cada macro bloco básico não correspondem fielmente às unidades funcionais físicas da arquitetura.

Desta forma, o *bypassing* adicionado entre os pares de unidades funcionais *físicas* deve ser mapeado para os pares de unidades funcionais *virtuais* de cada macro bloco básico. Por exemplo, a figura 5.10 mostra que, se $MAX\_BYPASSING$ for igual a 1, pelo menos as unidades funcionais *físicas* "$dp2" e "$dp1" terão *bypassing*. Entretanto, essas duas unidades funcionais físicas são mapeadas de formas diferentes entre os vários macro blocos básicos do programa. Assim, após determinarmos fisicamente quais os pares de unidades *físicas* terão *bypassing*, um mapeamento "reverso" desta informação deve ser realizado para cada as unidades *virtuais* de macro bloco básico. O algoritmo de inserção de cópias a ser executado no passo seguinte terá desta forma as informações necessárias para, possivelmente, negligenciar algumas instruções de cópia supostamente necessárias. Essa negligência é possível porque a rede de *bypassing* de operandos fará a passagem dos operandos necessários.

5.5.2 Inserção de Instruções de Cópias

Para cada dependência de dados entre operações que foram escalonadas para serem executadas em unidades funcionais diferentes, algumas questões precisam ser respondidas.

Primeiramente, verifica-se se as duas unidades funcionais correspondentes pertencem a um mesmo *cluster*. Neste caso, cópias não serão necessárias. Caso contrário, ou seja, caso as unidades funcionais pertençam a *clusters* distintos, verifica-se a existência de *bypassing* entre elas. Em caso afirmativo, há uma chance da instrução de cópia não ser necessária. Isso acontece quando a distância entre as operações dependentes é tal que o *bypassing* inserido possa ser efetivamente utilizado. Essa distância de escalonamento é calculada e, caso ultrapasse *nc* ciclos, a cópia é inserida[3]. Caso a distância entre as operações dependentes seja inferior a *nc*, a cópia não é inserida pois os operandos serão passados através da rede de *bypassing*.

Note que quando a cópia é negligenciada toma-se por base que a distância entre as operações *não é alterada* durante a execução do programa. Isso é verdade no caso de um escalonamento estático, a não ser pela ocorrência de eventuais interrupções e/ou exceções no processador. Portanto, para que o escalonamento de instruções proposto seja válido, os conteúdos dos registradores de *pipeline* também devem ser salvos nas trocas de contexto.

O algoritmo de inserção de cópias é repetido até que todas as instruções de cópias necessárias sejam inseridas. Note que cada instrução de cópia ocupa um *slot* de execução durante o período de um ciclo de relógio, no mesmo *slot* onde a operação que define o registrador foi executada. Assim, "buracos" necessitam ser abertos em algumas regiões do escalonamento para acomodar as instruções de cópia. Isso faz com que toda a *reservation_table* do macro bloco onde as cópias estão sendo inseridas seja constantemente alterado, até entrar em um regime permanente, onde o algoritmo pára. A convergência é garantida devido ao fato de que cada instrução terá que copiar o registrador que ela define para, no máximo, todas as outras instruções subseqüentes. Ainda, o escalonamento original define o *ASAP* de todas as operações, ou seja, com a inserção de cópias o ciclo inicial das instruções só pode aumentar ou permanecer o mesmo. A grosso modo, as operações "somente descem" na tabela a partir do ponto inicial.

A figura 5.12 ilustra um exemplo do funcionamento do algoritmo de inserção de cópias. Na figura, as tabelas representam o escalonamento para um macro bloco de um programa. Para simplificar o exemplo, suponha que o mapeamento das unidades funcionais virtuais é o mesmo da figura 5.7.a, ou seja, $dp1 = unidade virtual 0,... $dp4 = unidade virtual 3. Ainda, considere que arquitetura para este exemplo é a mostrada na figura 5.11.

[3] *nc* é o número máximo de ciclos que os operandos definidos pelas operações permanecem ativos nos registradores de *pipeline*. Num *datapath* com *pipeline* de 5 estágios, $nc = 3$.

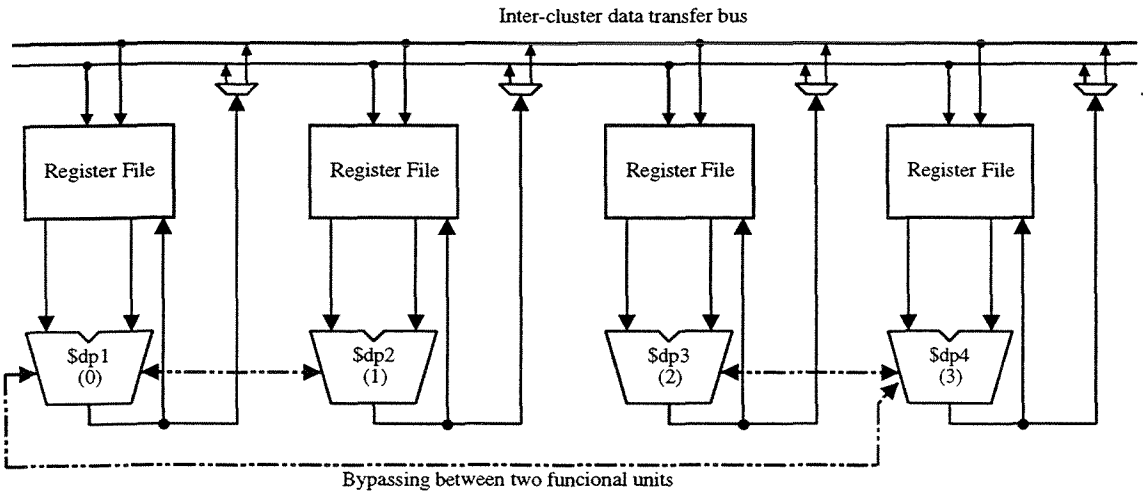


Figura 5.11: Arquitetura para o exemplo da figura 5.12.

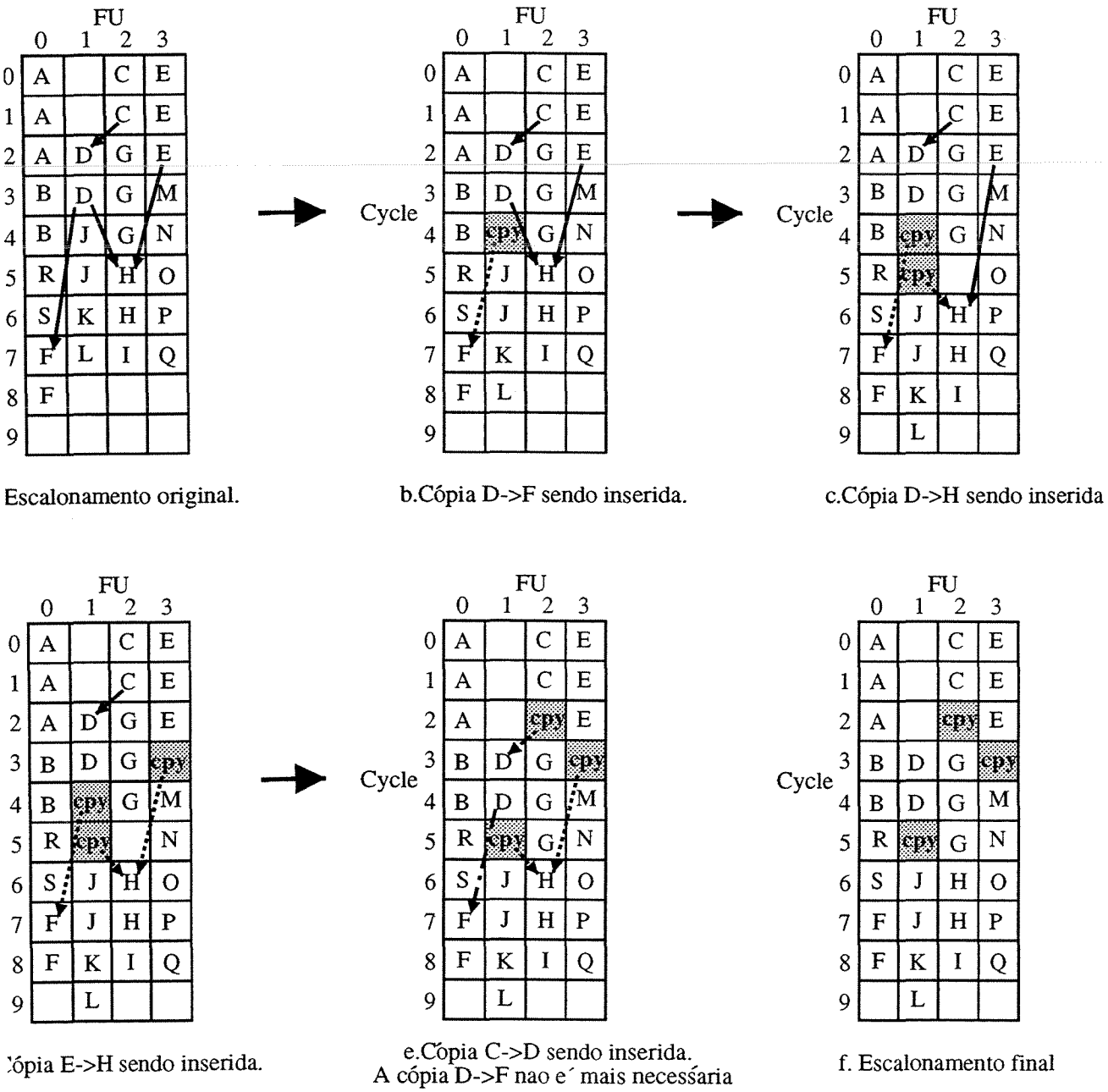


Figura 5.12: Exemplo do funcionamento do algoritmo de inserção de cópias.

A figura 5.12.a mostra o escalonamento original do macro bloco básico. As dependências de dados entre operações escalonadas em unidades funcionais diferentes são mostradas. Note que algumas destas dependências impõem a obrigatoriedade de uma instrução de cópia devido à falta de *bypassing* entre as unidades funcionais fonte e destino da dependência. Este é o caso, por exemplo, da dependência entre as operações "C" e "D". Observe na figura 5.11 que o par de unidades funcionais (1,2) não possui *bypassing* conectando-as. Outras dependências, entretanto, têm como fonte e destino pares de unidades ligadas por *bypassing*, aumentado assim a chance de não requererem cópias. Esse é o caso das dependências "E" $\rightarrow$ "H" e "D" $\rightarrow$ "F".

A figura 5.12.b mostra a inserção de uma instrução de cópia necessária para transferir um dos operandos requisitados pela operação "F". Observe que, mesmo havendo *bypassing* entre as unidades virtuais 1 e 0 o número de ciclos entre o final da operação "D" e o início da operação "F" é maior do que o número máximo *nc* permitido para a utilização de uma comunicação direta (3).

A figura 5.12.c ilustra outra instrução de cópia sendo inserida, neste caso para resolver a dependência de dados entre as operações "D" e "H". Neste caso, mesmo as operações estando próximas, não há *bypassing* de operandos entre as unidades funcionais 1 e 2, ou seja, a cópia é imprescindível.

A figura 5.12.d apresenta mais uma inserção de cópia. Neste caso, as duas operações dependentes são "E" e "H". Semelhantemente ao primeiro caso, mesmo havendo *bypassing* entre as unidades 3 e 2 o número de ciclos que separa as duas operações ultrapassa o máximo estabelecido.

A figura 5.12.e mostra a inserção da instrução de cópia para resolver a dependência entre as operações "C" e "D". Este ponto é particularmente interessante pois note que instrução de cópia anteriormente inserida para resolver a dependência entre "D" e "F" torna-se não necessária. Um dos efeitos da inserção da cópia para resolver "C" e "D" foi diminuir a distância entre as operações "D" e "F", colocando-a na faixa onde o *bypassing* entre as unidades 0 e 1 possa ser utilizado. Observe ainda que o *overhead* introduzido pelas instruções de cópia é de 1 ciclo, ou seja, 11.1%. Isto provavelmente poderá ser compensado por uma redução no *cycle time* do processador devido a bancos de registradores menores.

Finalmente, a figura 5.12.f mostra o escalonamento final do macro bloco básico com todas as instruções de cópias necessárias.

5.6 Inserção de NOPs

Quando duas operações dependentes são escalonadas em duas unidades funcionais que pertencem a um mesmo *cluster* nenhuma instrução de cópia é necessária. Entretanto, se as decisões de particionamento foram tais que estas unidades funcionais não possuem

bypassing de operandos, e a distância entre as duas operações dependentes é menor ou igual a dois ciclos de relógio[4] um *data hazard* ocorre e portanto uma (ou duas) instruções de NOP deve(m) ser inserida(s) pelo compilador a fim de resolver o *hazard*. Os NOPs são inseridos na unidade funcional, e a partir do ciclo, onde a operação destino da dependência estava alocada. Desta forma, o novo ciclo inicial para a operação destino é calculado como sendo o ciclo inicial original mais o número de NOPs necessários. A figura 5.13 mostra as instruções de NOPs sendo inseridas em dois exemplos. Em um dos casos a distância original entre as operações é de um ciclo, exigindo portanto dois NOPs. No outro, essa distância é de dois ciclos, e portanto apenas um NOP é necessário.

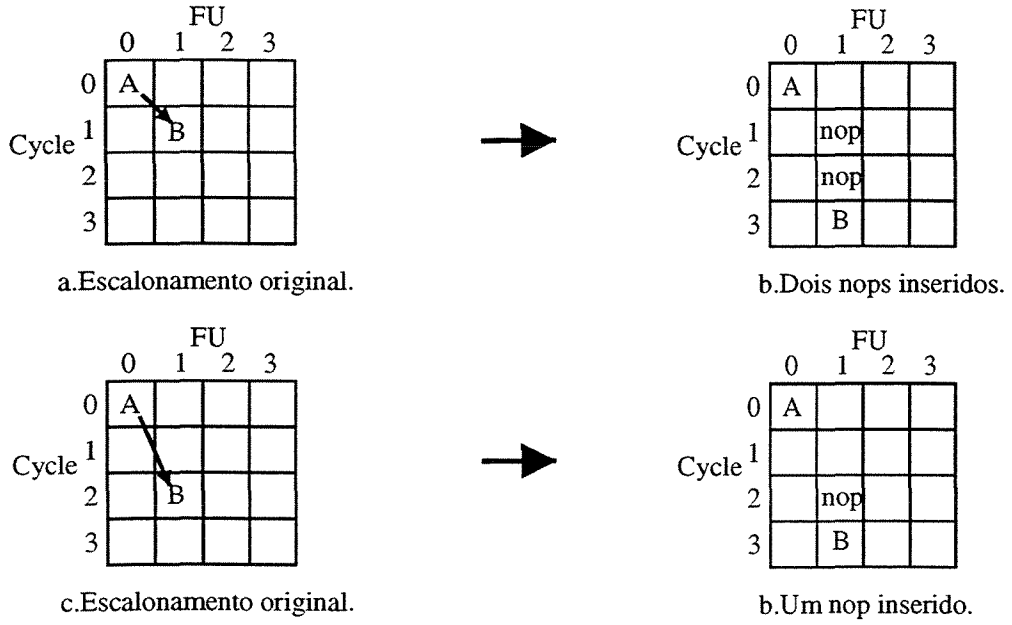


Figura 5.13: Exemplo do funcionamento da inserção de nops.

[4]Para um *pipeline* RISC de 5 estágios

5.7 Cópias Redundantes

Outra característica do algoritmo de inserção cópias é o de não inserir cópias redundantes. Por exemplo, considere o caso em que duas operações sendo executadas em duas unidades funcionais distintas, "2" e "3", que pertencem a um mesmo *cluster* , "CL2", necessitam de um operando, "r1", definido por uma terceira operação sendo executada em uma unidade funcional "0", localizada em um segundo *cluster* ,"CL1". Neste caso, se as duas operações em "2" e "3" necessitarem de instruções de cópia, apenas uma delas é inserida. A partir deste ponto, as duas operações têm acesso ao registrador requerido através do banco de registradores local compartilhado pelas mesmas. A figura 5.14 ilustra o cenário descrito . Suponha na figura que a arquitetura considerada possui 4 unidades funcionais distribuídas em dois *clusters*. As unidades funcionais virtuais "0" e "1" formam o ıcluster "CL1" e as unidades "2" e "3" formam o *cluster* "CL2".

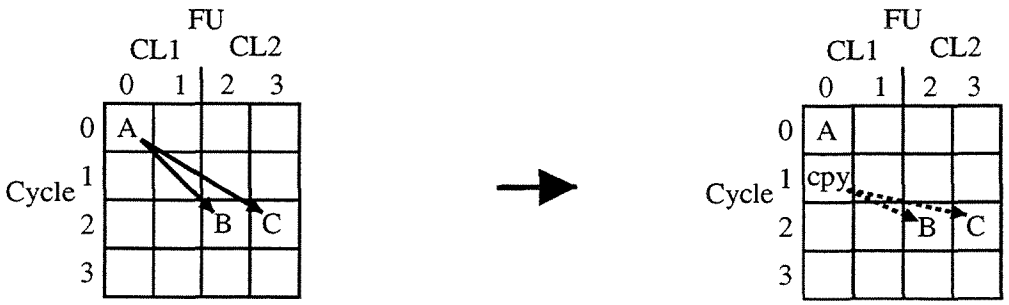


Figura 5.14: Cópias redundantes.

No caso de uma das unidades funcionais, "2" ou "3", possuir *bypassing* de operandos com a unidade "0", e a distância entre as operações dependentes for tal que o *bypassing* possa ser utilizado, o número de cópias continua sendo 1. No caso das duas unidades "2" e "3" possuírem *bypassing* com a unidade "0", e as duas distâncias permitirem o uso de *bypassing*, nenhuma cópia é gerada.

5.8 Número máximo de cópias por ciclo

Um ponto chave a ser considerado pelo algoritmo de inserção de cópias diz respeito ao número máximo de instruções de cópias permitidas em cada ciclo de relógio. Como o objetivo principal do particionamento da arquitetura VLIW em vários bancos de registradores é diminuir o número de portas de cada banco, o número de barramentos especiais

destinados à passagem de operandos via instrução de cópia deve ser minimizado. Essa minimização limita o número máximo de instruções de cópia permitidas por ciclo de relógio. Uma heurística retirada de [8] foi utilizada com o objetivo de calcular um número de barramentos de cópia que fosse coerente com o objetivo de minimizar o número de portas por banco e ainda assim prover uma comunicação aceitável entre *clusters*.

Desta forma, a expressão abaixo define o número máximo de instruções de cópia permitidas em cada ciclo de relógio implementado no algoritmo:

$$(Numero\_de\_clusters - 1) * \frac{Numero\_de\_unidades\_funcionais/Numero\_de\_clusters}{2}$$

Assim sendo, se no momento de inserção de uma cópia o número máximo permitido já tenha sido atingido, NOPs são inseridos para atrasar a cópia até que um ciclo onde a mesma possa ser alocada seja encontrado.

5.9 Resultados Experimentais

5.9.1 Considerações Iniciais

Com a finalidade de avaliar a eficiência do algoritmo proposto neste capítulo, uma implementação do mesmo foi realizada tendo-se como infraestrutura o compilador IMPACT [32]. IMPACT é um compilador otimizante voltado a arquiteturas paralelas, como superescalar e VLIW. Várias técnicas de aumento do volume de paralelismo são utilizadas por este compilador. *Loop-unrolling*, formação de super-blocos, formação de hiper-blocos, construção do PDG, execução predicada, são alguns exemplos das técnicas utilizadas.

Os *benchmarks* avaliados foram SPECInt 95, SPECFp 95 e o conjunto *Miscelaneuos* descritos nos capítulos anteriores.

Várias configurações de arquitetura foram testadas no sentido de se estimar o número de ciclos gastos para executar as aplicações em cada uma delas. Os parâmetros que variam de uma configuração para outra são o número de unidades funcionais, o número de bancos de registradores e o "volume" da rede de *bypassing* de operandos permitido ($MAX\_BYPASSING$). Uma, duas, quatro, oito e dezesseis unidades funcionais foram consideradas, assim como um, dois, quatro, oito e dezesseis bancos de registradores. Logicamente, somente as combinações consideradas "válidas" foram utilizadas. Por exemplo, uma arquitetura com 4 unidades funcionais e 8 bancos de registradores não apresenta lógica alguma. Ainda, o parâmetro $MAX\_BYPASSING$ variou desde 1 até 16, novamente observando-se a validade de cada combinação FU-Banco-*bypassing*.

Rótulos identificam cada uma das configurações sendo testadas. A figura 5.15 mostra como os rótulos são definidos.

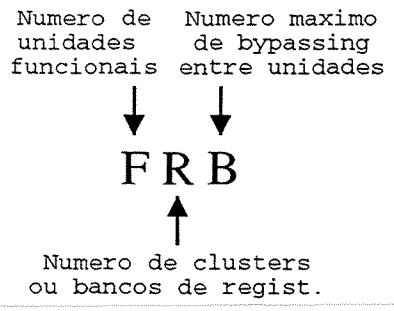


Figura 5.15: Definição dos rótulos de cada configuração de arquitetura.

Desta forma, o rótulo "421" representa, por exemplo, uma arquitetura com quatro unidades funcionais, dois bancos de registradores e somente um par de unidades funcionais, no máximo, com mecanismo de *bypassing* (portanto, o par unidades mais comunicante).

5.9.2 Análise dos Resultados

As figuras 5.16 e 5.17 mostram o número de ciclos gastos por todas as 67 configurações testadas para o programa *compress*. Na figura 5.16 são mostradas as configurações desde "110", representando uma arquitetura escalar, até "888", representando uma arquitetura com 8 unidades funcionais, 8 bancos de registradores e no máximo 8 pares de unidades contendo *bypassing*. Na realidade, este gráfico poderia ser dividido em quatro regiões, cada uma representando o subconjunto das configurações de um número fixo de unidades funcionais. Por exemplo, todas as configurações desde "210" até "221" representam as variações possíveis quando consideram-se apenas duas unidades funcionais. Da mesma forma, o intervalo entre "410" e "444" expõe as variantes arquiteturais com número de FUs igual a 4.

Na figura 5.17 são mostradas as diferentes configurações considerando-se 16 unidades funcionais. Note que o eixo das abscissas neste último gráfico mostra somente dois números para cada configuração, o primeiro representando o número de bancos de registradores e o segundo o número máximo de *bypassing* permitido. Esta escolha foi em prol de uma maior clareza da figura, já que o número de unidades funcionais é sempre 16.

Desta maneira, observe o comportamento dos pontos "*(configuração, número de ciclos)*" na figura 5.16. Um comportamento similar aparece em cada uma das subregiões, assim como nas divisas entre elas. Isto é, quando o número de unidades funcionais aumenta, considerando-se um número fixo de bancos de registradores e de *bypassing*, o número de ciclos geralmente diminui. Exemplos: ("210", "410" e "810") ; ("211", "411" e "811") ; ("220", "420" e "820") ; ("221", "421" e "821") ; ("412" e "812") ; ("414" e "814") ;

etc. A justificativa ara este fato é intuitiva, uma vez que mais unidades funcionais significa mais paralelismo útil na arquitetura.

Dentro de uma mesma região (número de unidades funcionais constante), mais bancos de registradores significa mais instruções de cópia sendo necessárias. Assim, se considerarmos constante também o volume de *bypassing*, o número de ciclos geralmente aumenta à medida que o número de *clusters* cresce. Exemplos: ("410", "420" e "440") ; ("411", "421" e "441") ; ("412", "422" e "442") ; ("414", "424" e "444").

Por último, mantendo-se o número de unidades funcionais e o número de bancos de registradores constantes, tem-se uma redução do número de ciclos à medida que adiciona-se mais *bypassing* à arquitetura. Como esperado, a distância entre a definição e o uso das variáveis é pequena, fazendo com que a adição de mecanismos de *bypassing* (mesmo que entre unidades funcionais de *clusters* diferentes) auxilie na redução do número de ciclos para executar uma aplicação. Exemplos: ("410", "411", "412" e "414") ; ("420", "421", "422" e "424") ; ("440", "441", "442" e "444").

Note que os resultados apresentados levam a uma conclusão muito interessante: o número de ciclos gastos por configurações com muitas unidades funcionais e poucas linhas de *bypassing* é às vezes semelhante ao número de ciclos utilizados por configurações com menos unidades funcionais porém com um volume razoável de *bypassing* entre elas. Isso significa que, ao invés de aumentar-se o número de unidades funcionais quando deseja-se mais desempenho, especializar a arquitetura através da inserção cautelosa de linhas de *bypassing* entre as unidades mais comunicantes pode resolver o problema.

Outra bateria de resultados para o programa *compress* é apresentada na figura 5.18. Em cada uma das 4 partes que compõem esta figura são apresentados resultados relativos a um dos subconjuntos de configurações citadas anteriormente. O número de unidades funcionais N_FUs é constante em cada sub-figura e o número de bancos de registradores varia desde 1 até N_FUs. Ainda, várias curvas são mostradas correspondendo aos diferentes valores de $MAX\_BYPASSING$ permitidos.

O panorama geral apresentado mostra um cenário onde aumentando-se o número de bancos de registradores cresce também o número de ciclos para executar o programa. Esse resultado é esperado uma vez que instruções de cópia degradam o desempenho da aplicação. Entretanto, note que em alguns casos o acréscimo no número de bancos de registradores às vezes conduz a um menor número de ciclos. Por exemplo, a curva representando "00" *bypassing* na figura 5.18.c mostra uma queda de ciclos quando passa de 1 para 2 bancos de registradores. Aparentemente este resultado causa estranheza, visto que o único banco centralizado de registradores foi dividido em dois bancos. Todavia, note também que o $MAX\_BYPASSING$ considerado é zero, ou seja, as unidades funcionais que compartilham o banco centralizado comunicam-se unicamente pelo próprio banco. Desta forma, NOPs preenchendo até *dois* ciclos devem ser inseridos para resolver certas

dependências de dados entre operações próximas. Quando o número de bancos de registradores passa para 2, algumas das resoluções de dependências passam a ser feitas via instrução de cópia, que gasta apenas 1 ciclo. Portanto, certas vezes a divisão do banco de registradores resulta em um menor número de ciclos porque 1 cópia é mais barata do que 2 NOPs.

Para todos os programas também são mostrados gráficos referentes ao número de cópias obtido nas várias configurações de arquitetura testadas. Note porém que o número de cópias apresentado nos gráficos se refere ao número absoluto de cópias, isto é, não ponderado com relação ao nível de aninhamento dos laços (*nesting level*). O que está sendo mostrado, portanto, é o número real de instrções de cópias que foram adicionadas ao programa, e não a quantidade de vezes que as instruções de cópias serão executadas.

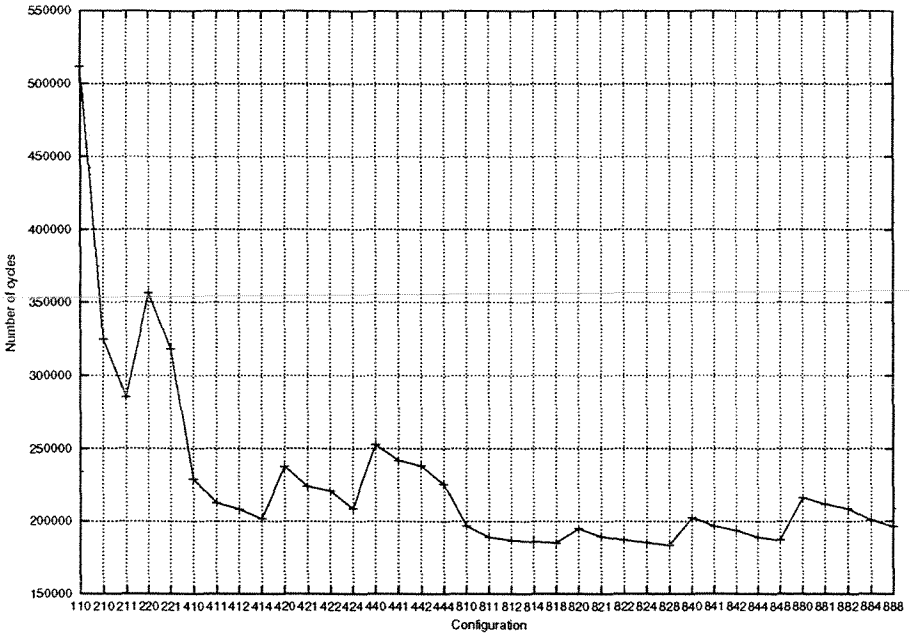


Figura 5.16: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *compress*.

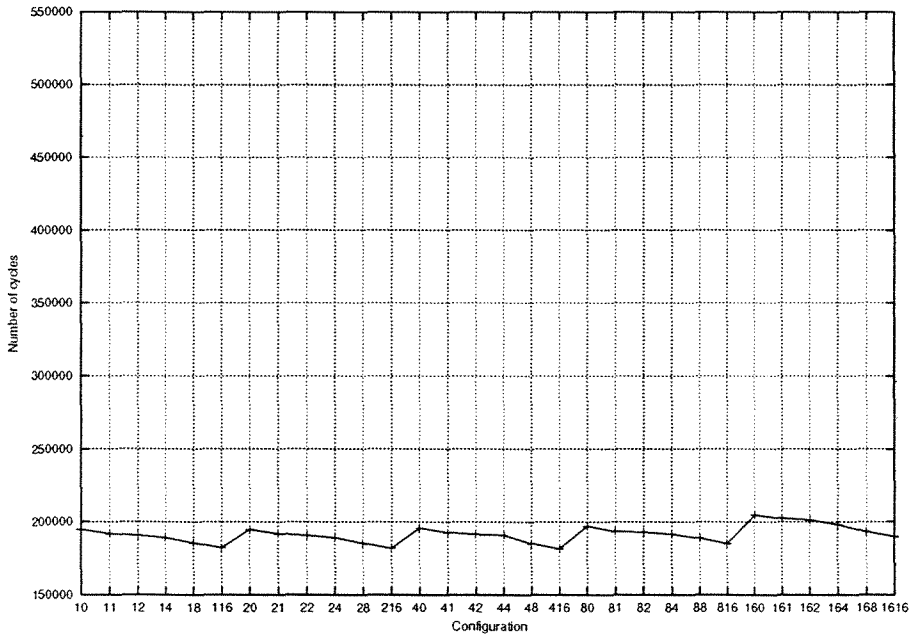
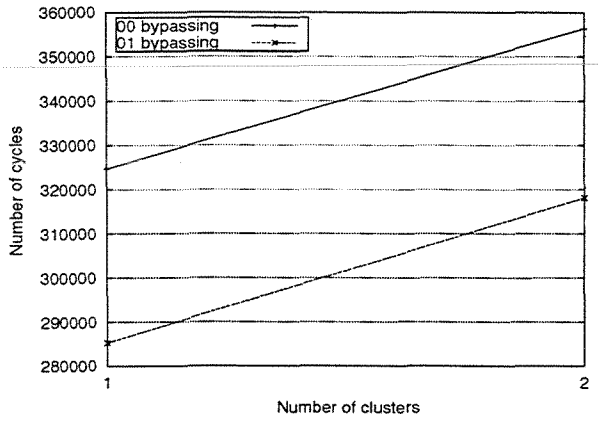
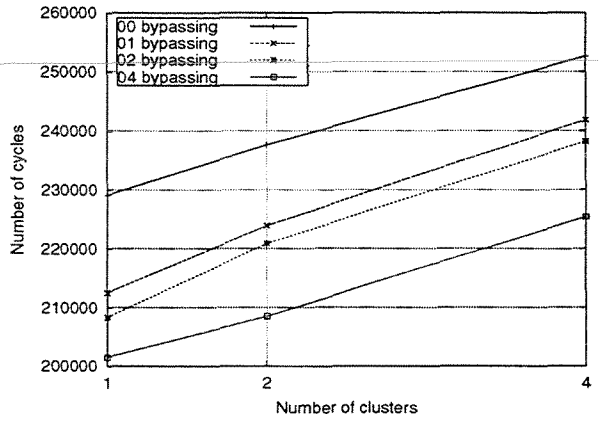


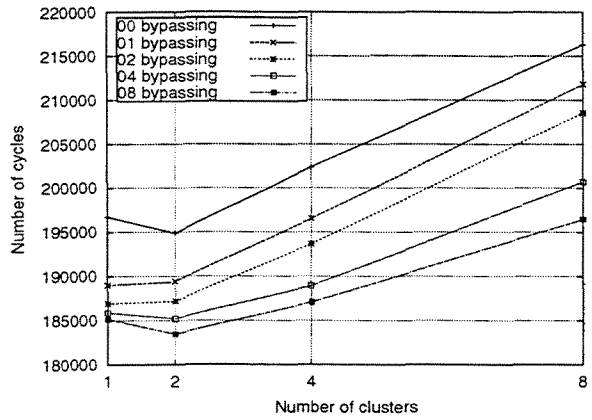
Figura 5.17: Número de ciclos gastos pelas configurações de 16 FUs para o programa *compress*.



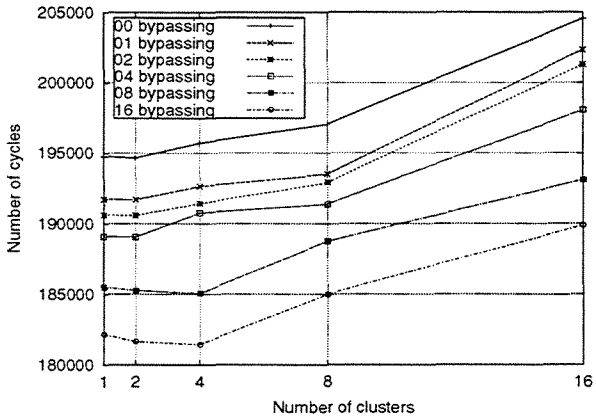
(a) 2 FUs



(b) 4 FUs

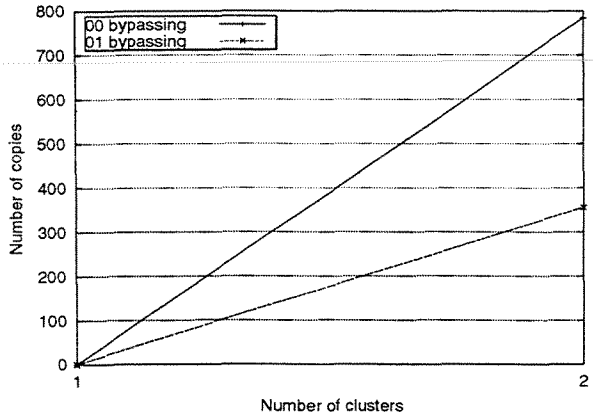


(c) 8 FUs

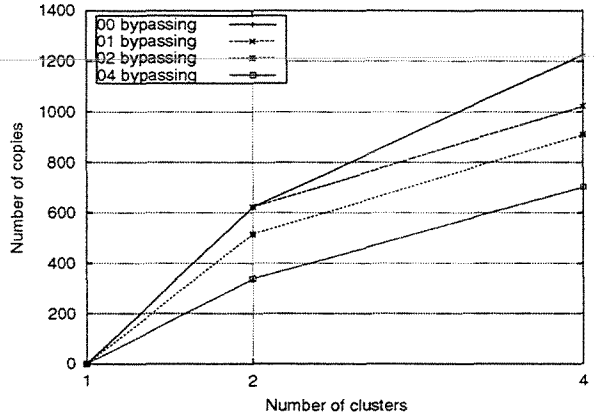


(d) 16 FUs

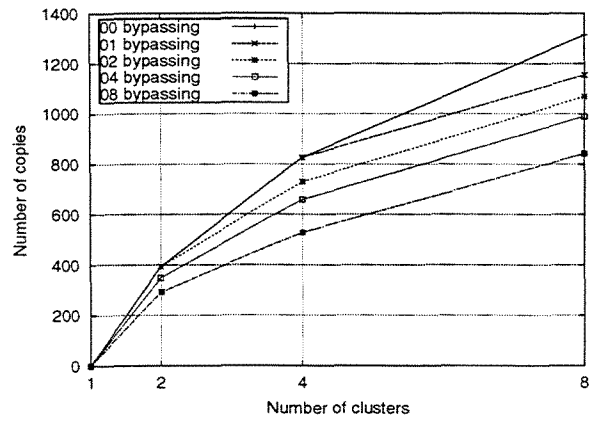
Figura 5.18: Resultados agrupados para o programa *compress* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



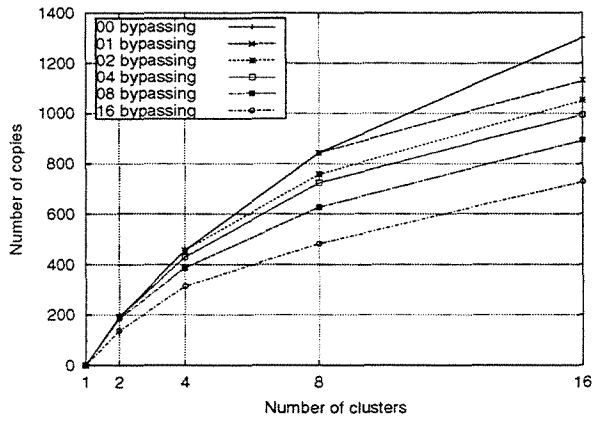
(a) 2 FUs



(b) 4 FUs



(c) 8 FUs



(d) 16 FUs

Figura 5.19: Número de instruções de cópia para o programa *compress* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

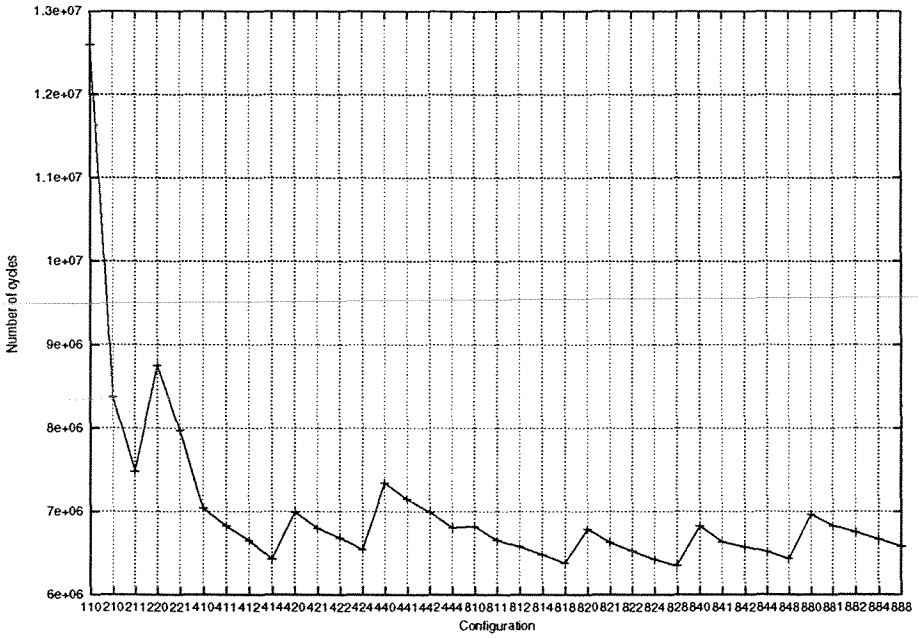


Figura 5.20: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *go*.

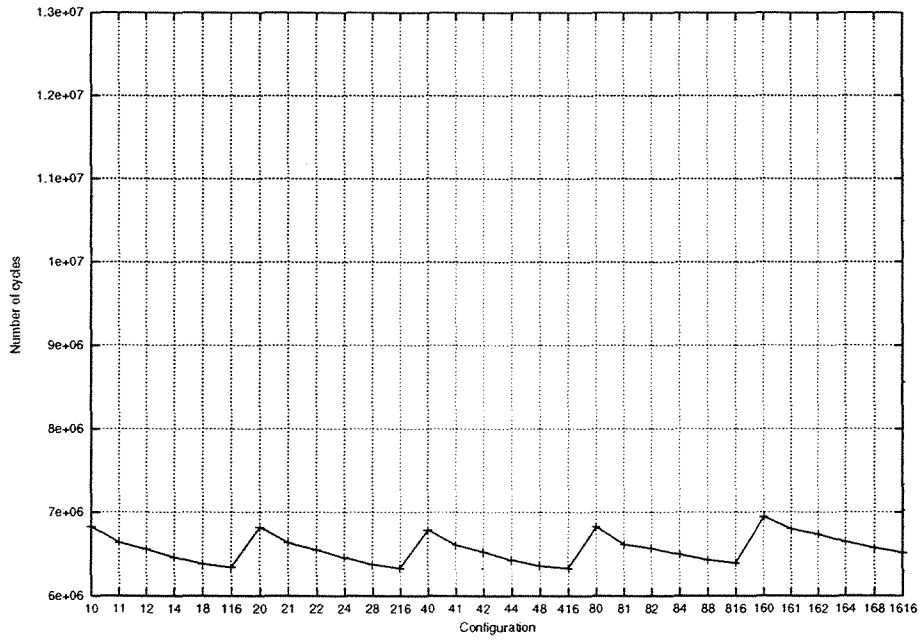
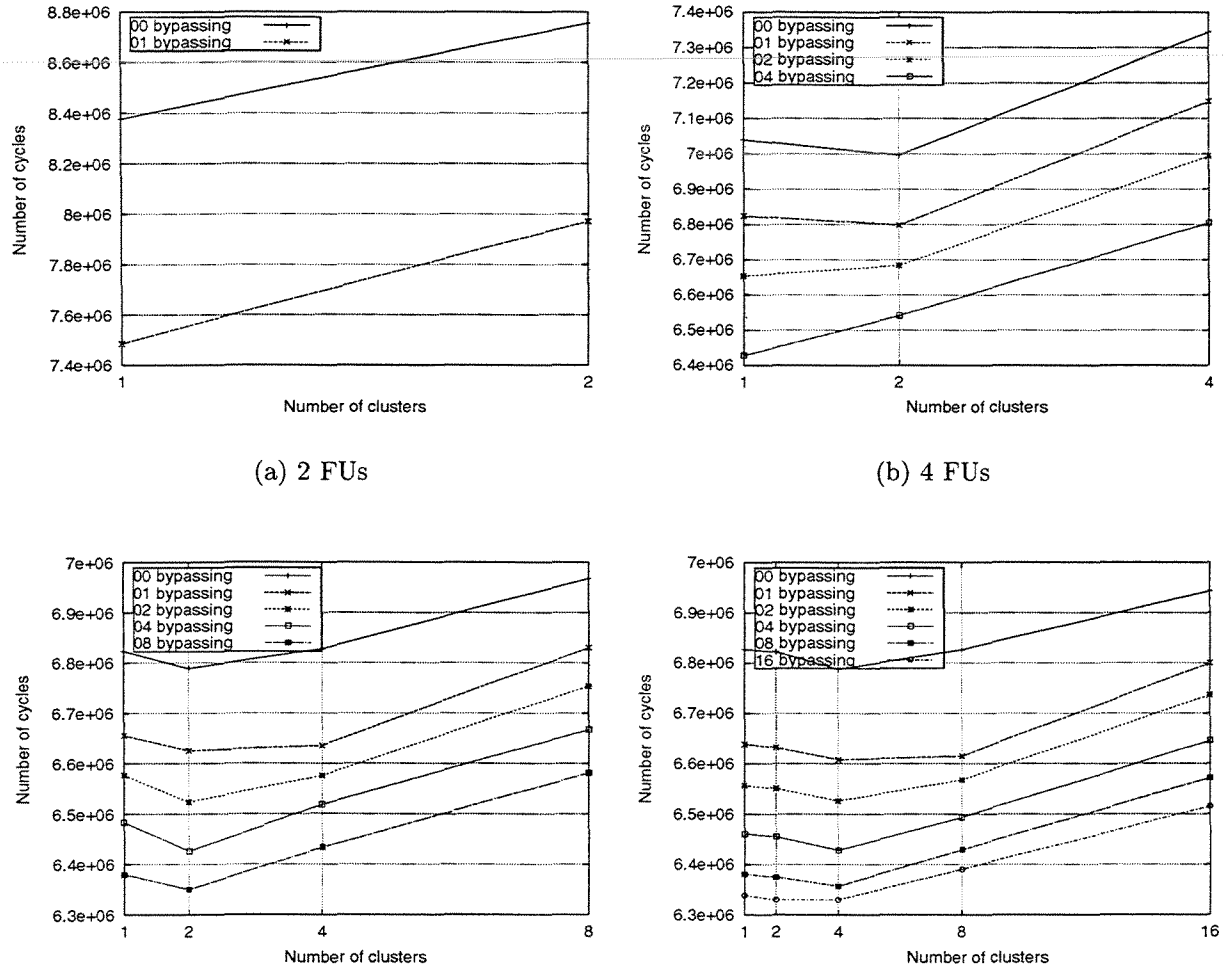


Figura 5.21: Número de ciclos gastos pelas configurações de 16 FUs para o programa *go*.



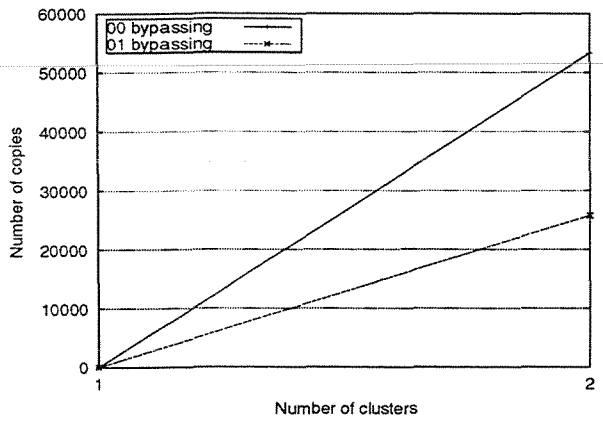
(a) 2 FUs

(b) 4 FUs

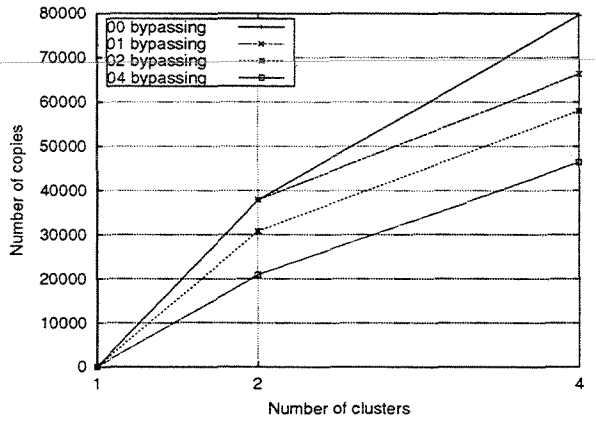
(c) 8 FUs

(d) 16 FUs

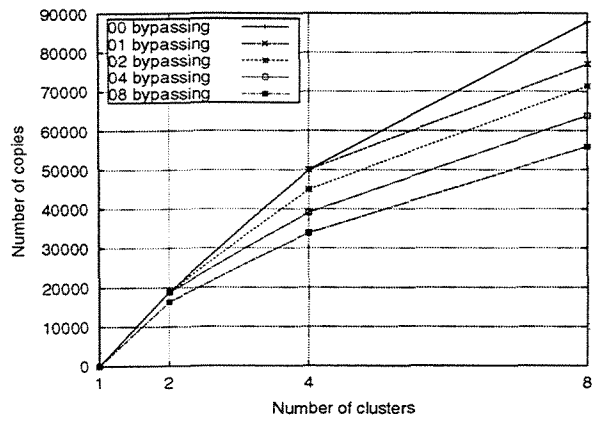
Figura 5.22: Resultados agrupados para o programa *go* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



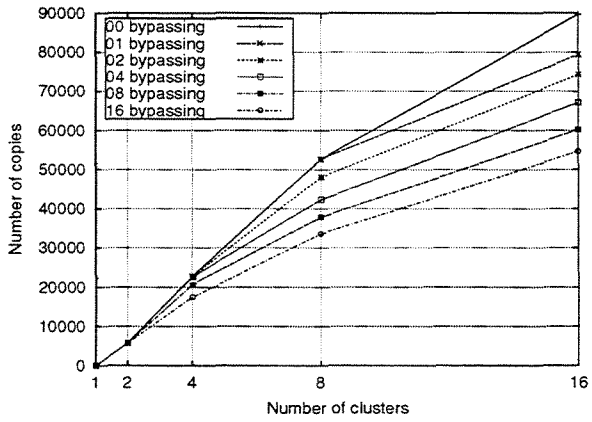
(a) 2 FUs



(b) 4 FUs



(c) 8 FUs



(d) 16 FUs

Figura 5.23: Número de instruções de cópia para o programa *go* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

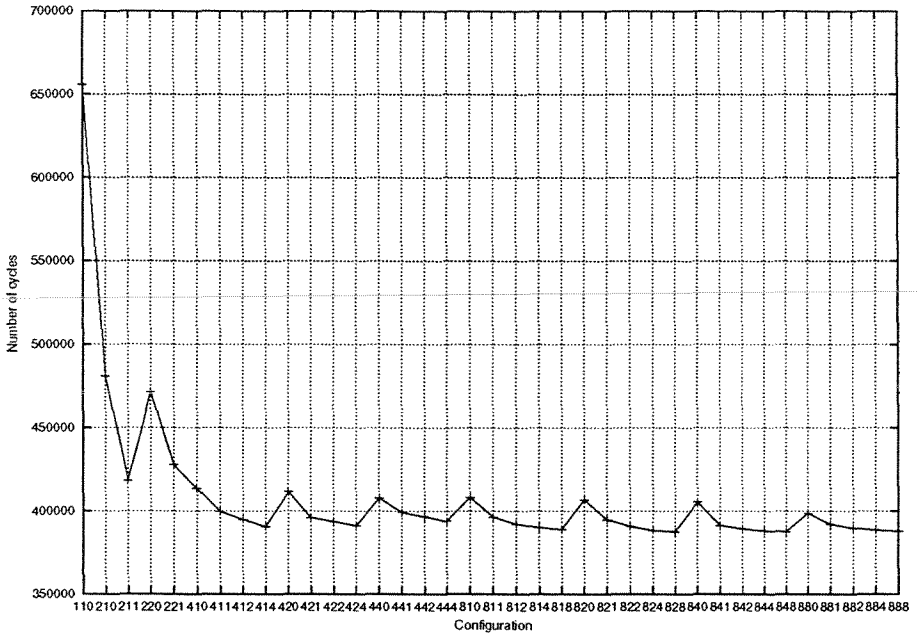


Figura 5.24: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *m88ksim*.

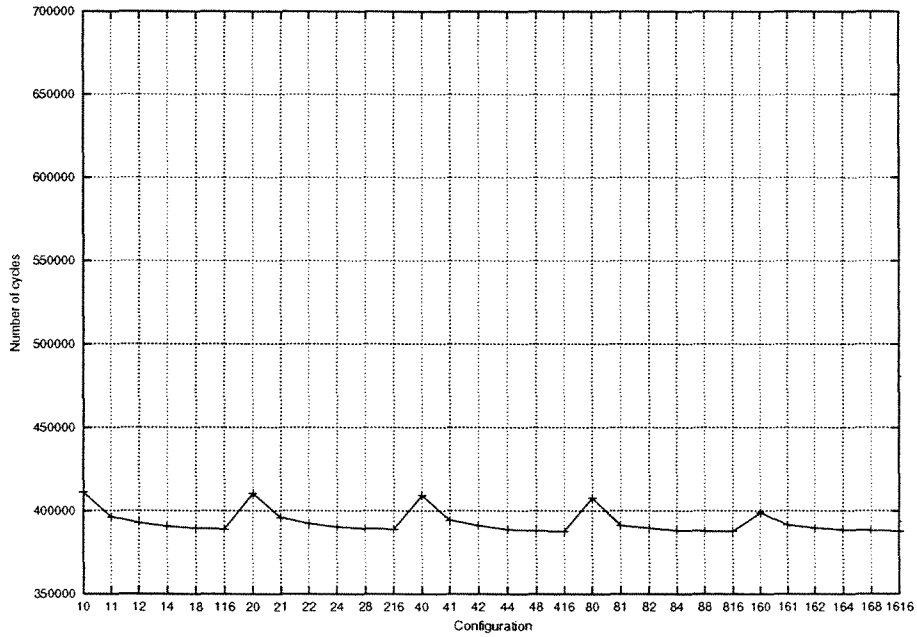
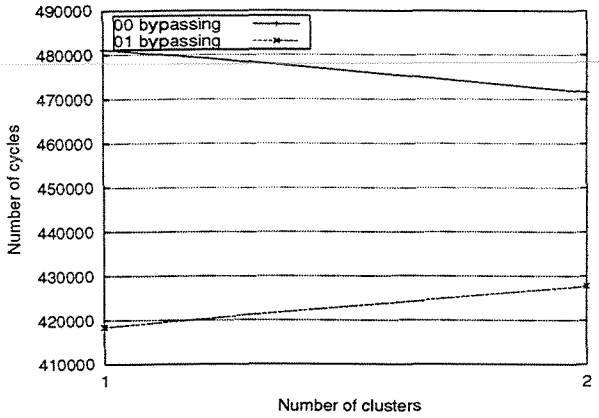
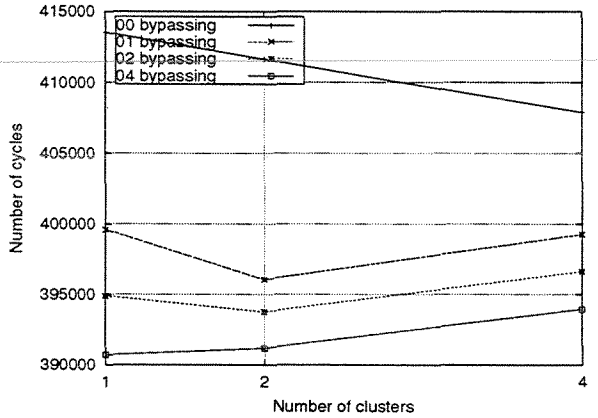


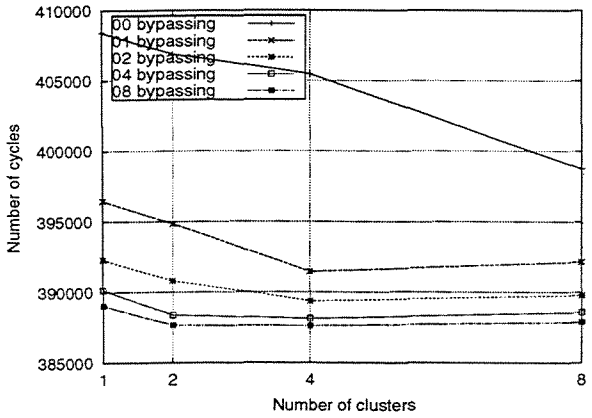
Figura 5.25: Número de ciclos gastos pelas configurações de 16 FUs para o programa *m88ksim*.



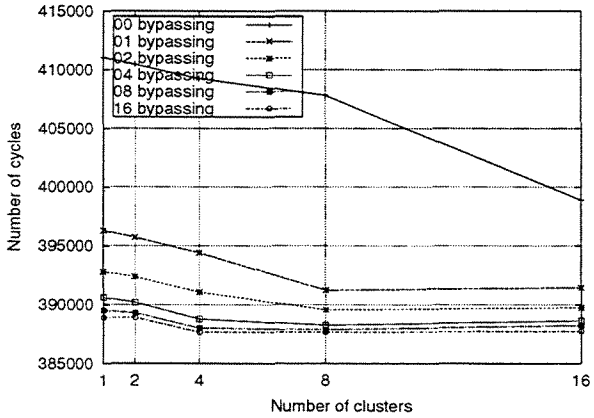
(a) 2 FUs



(b) 4 FUs

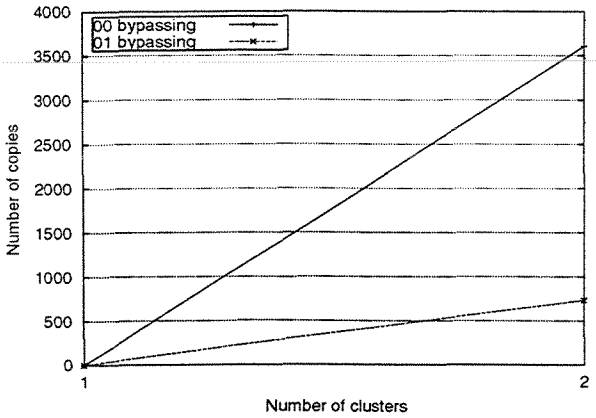


(c) 8 FUs

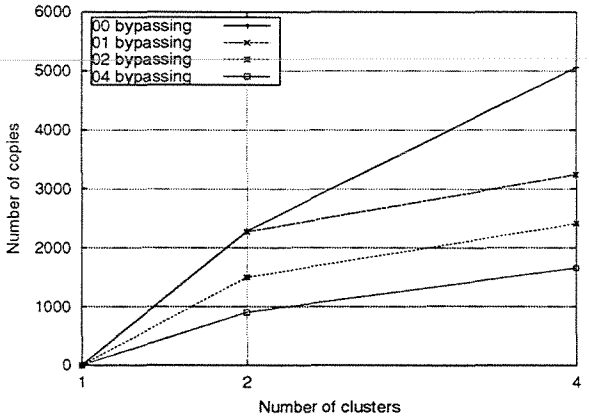


(d) 16 FUs

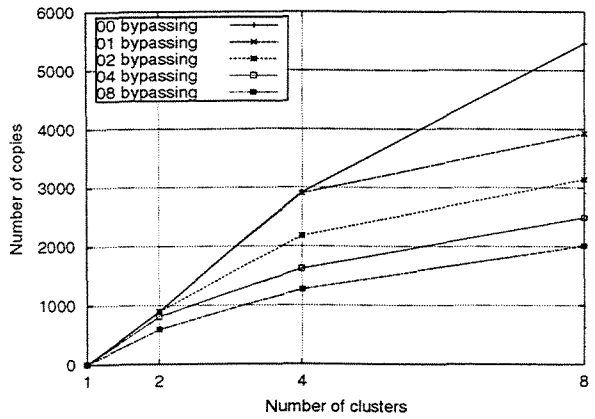
Figura 5.26: Resultados agrupados para o programa *m88ksim* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



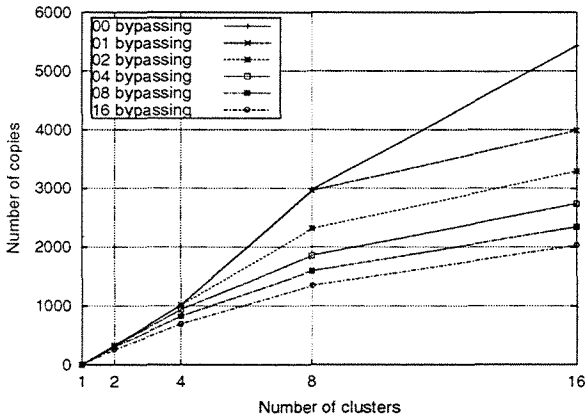
(a) 2 FUs



(b) 4 FUs



(c) 8 FUs



(d) 16 FUs

Figura 5.27: Número de instruções de cópia para o programa *m88ksim* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

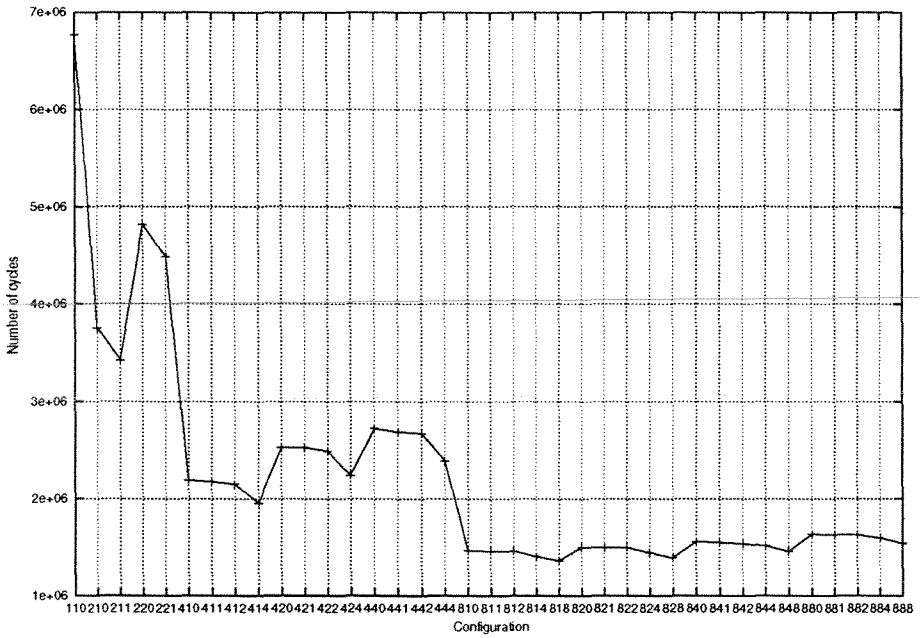


Figura 5.28: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *mgrid*.

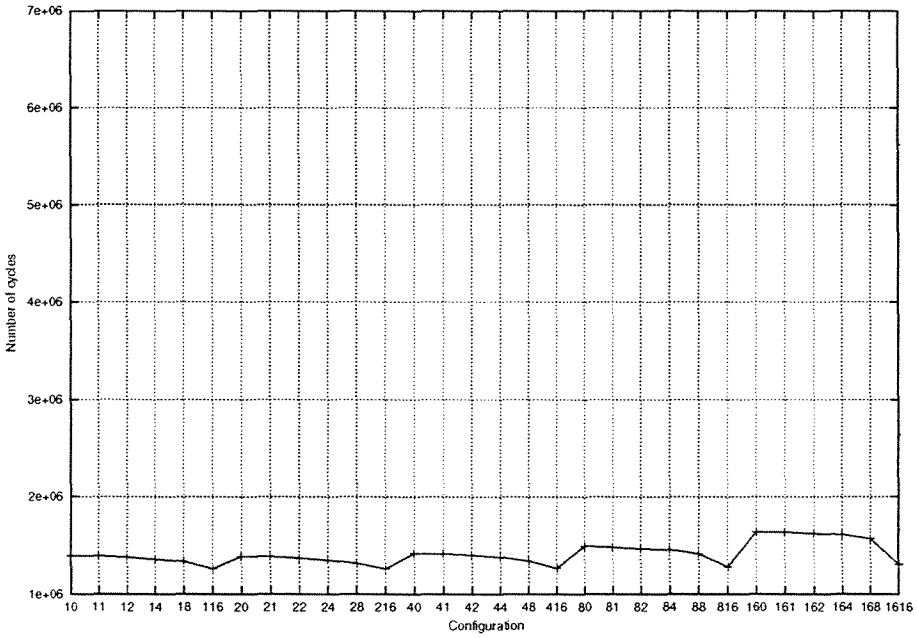
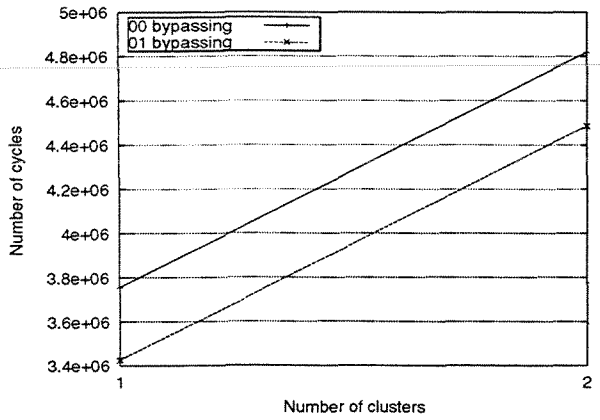
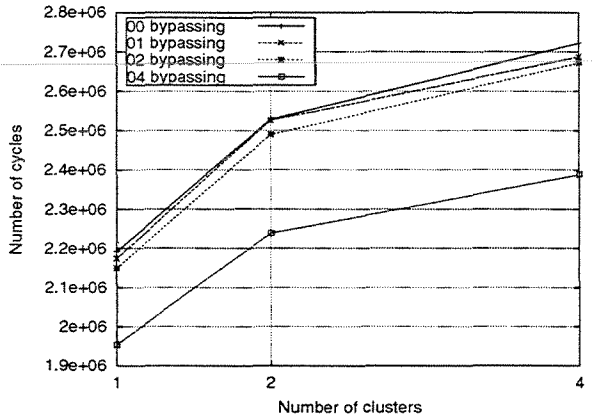


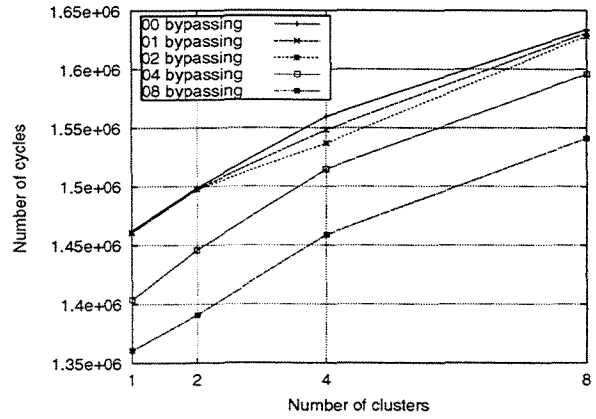
Figura 5.29: Número de ciclos gastos pelas configurações de 16 FUs para o programa *mgrid*.



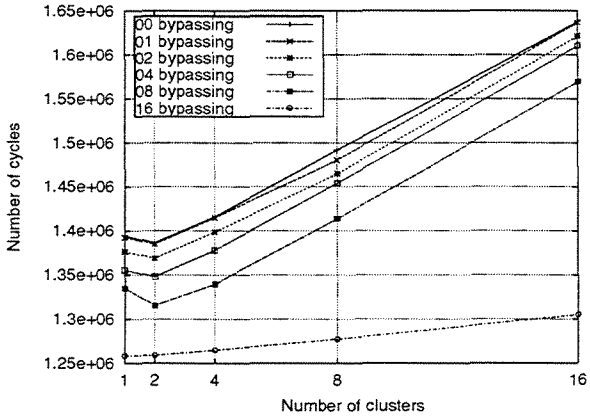
(a) 2 FUs



(b) 4 FUs

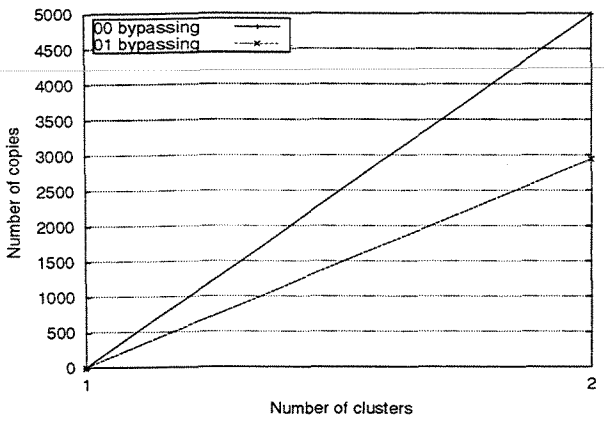


(c) 8 FUs

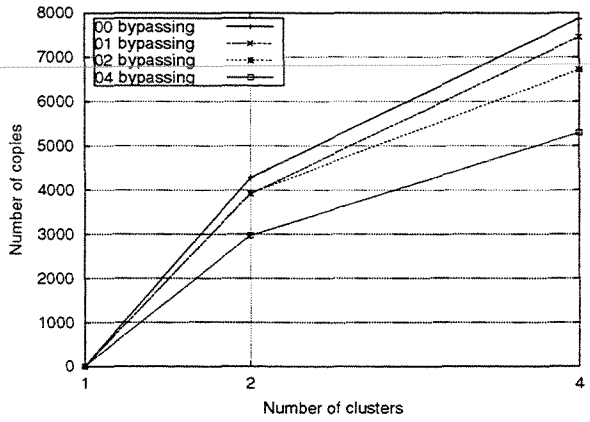


(d) 16 FUs

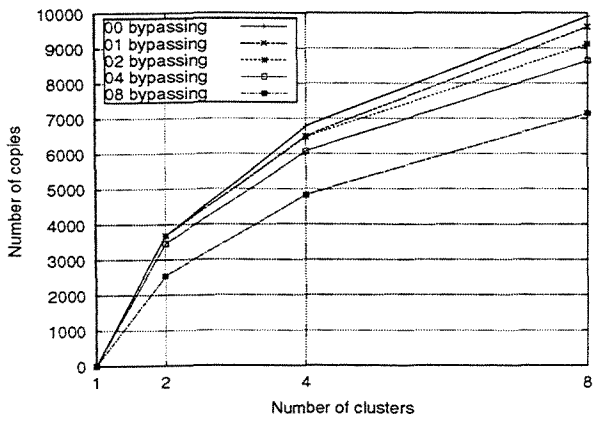
Figura 5.30: Resultados agrupados para o programa *mgrid* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



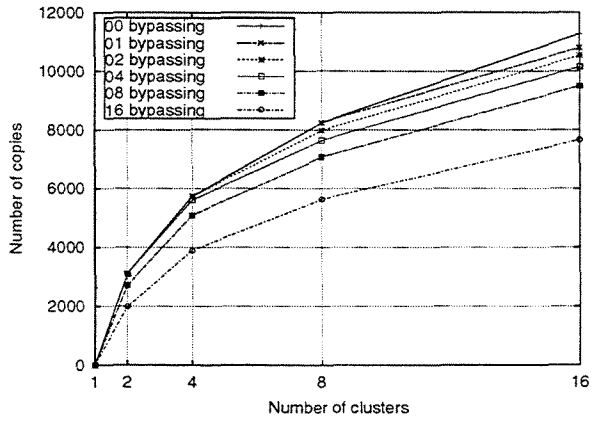
(a) 2 FUs



(b) 4 FUs



(c) 8 FUs



(d) 16 FUs

Figura 5.31: Número de instruções de cópia para o programa *mgrid* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

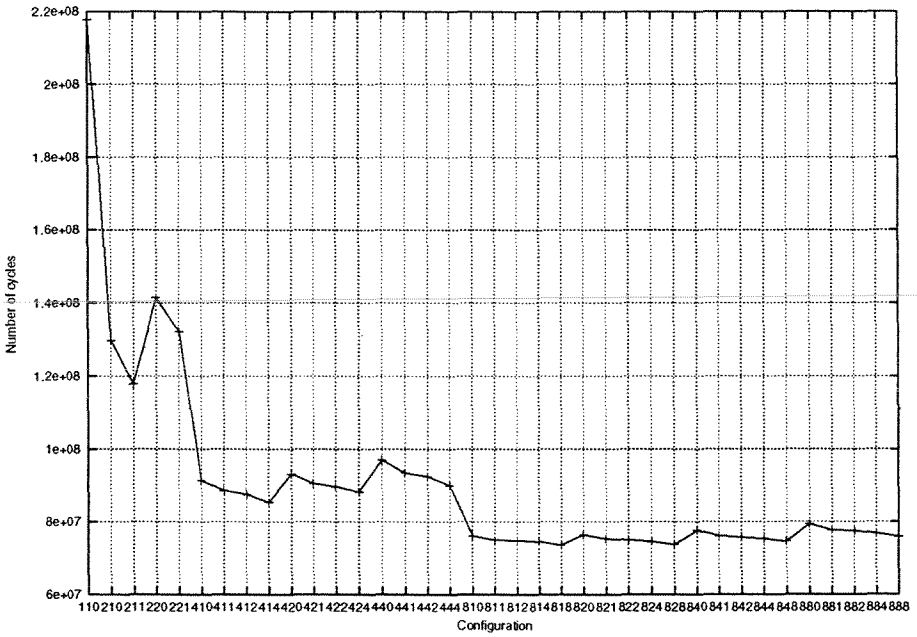


Figura 5.32: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *su2cor*.

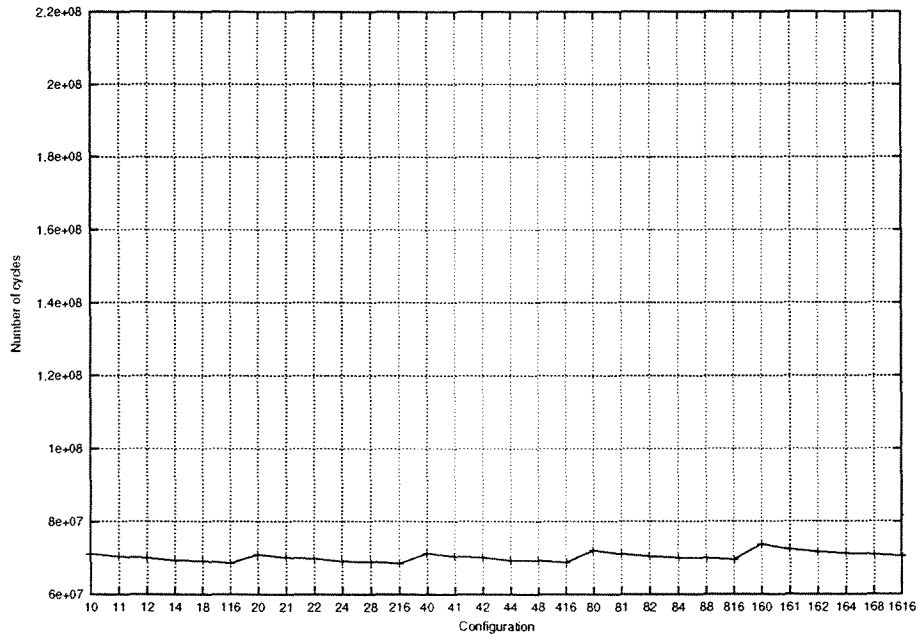
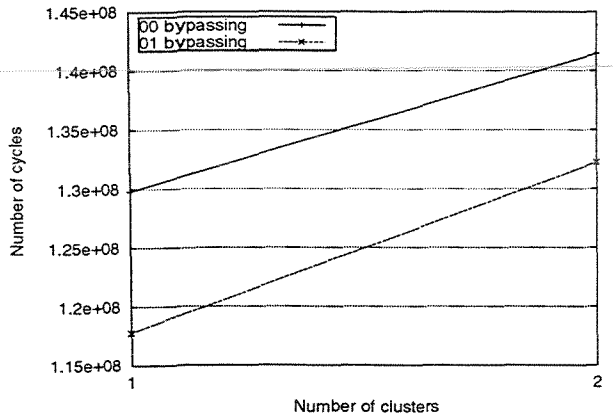
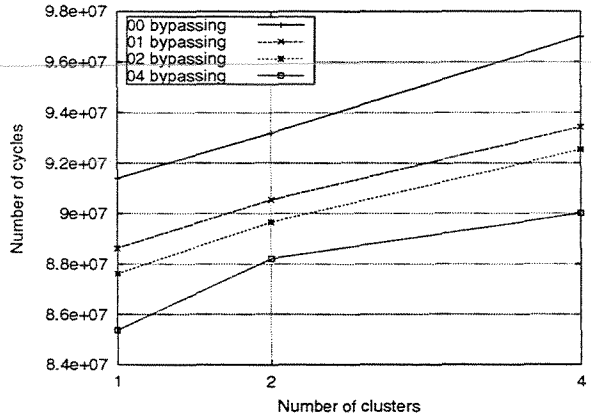


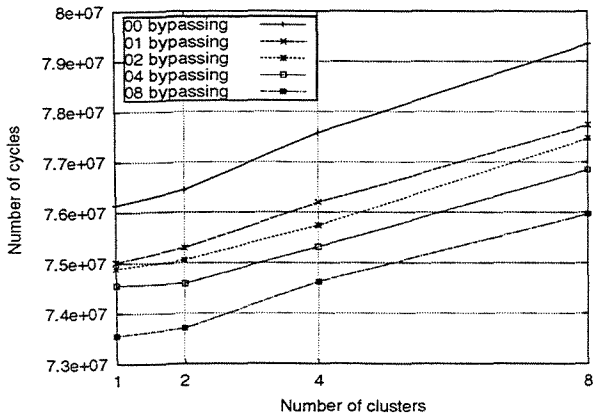
Figura 5.33: Número de ciclos gastos pelas configurações de 16 FUs para o programa *su2cor*.



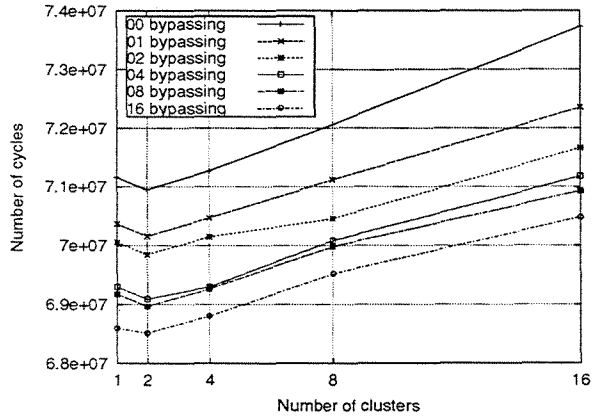
(a) 2 FUs



(b) 4 FUs

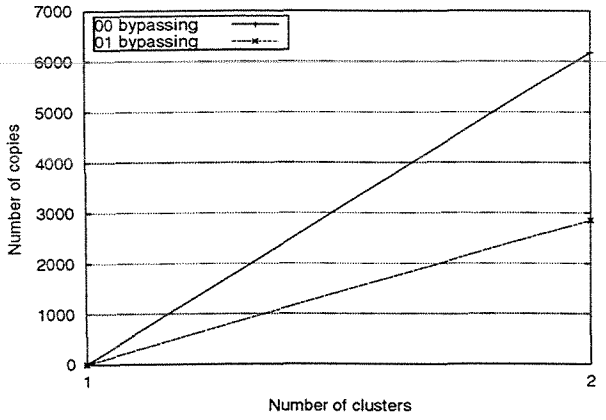


(c) 8 FUs

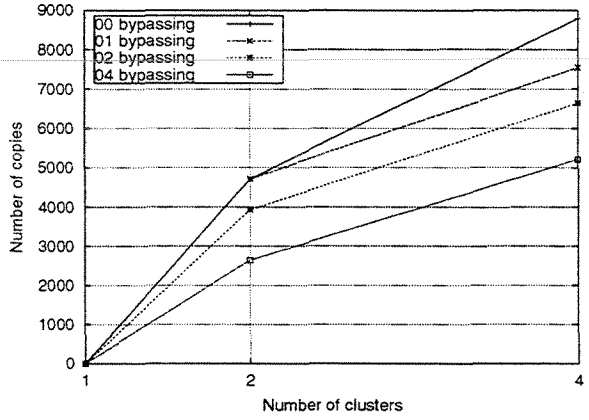


(d) 16 FUs

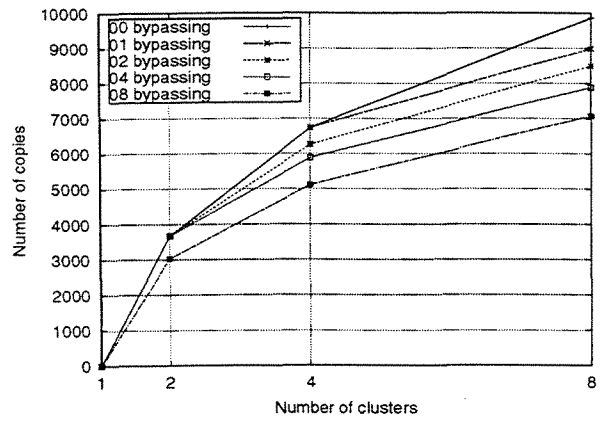
Figura 5.34: Resultados agrupados para o programa *su2cor* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



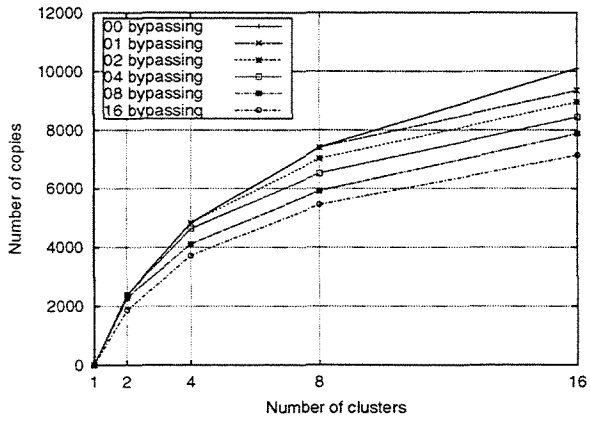
(a) 2 FUs



(b) 4 FUs



(c) 8 FUs



(d) 16 FUs

Figura 5.35: Número de instruções de cópia para o programa *su2cor* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

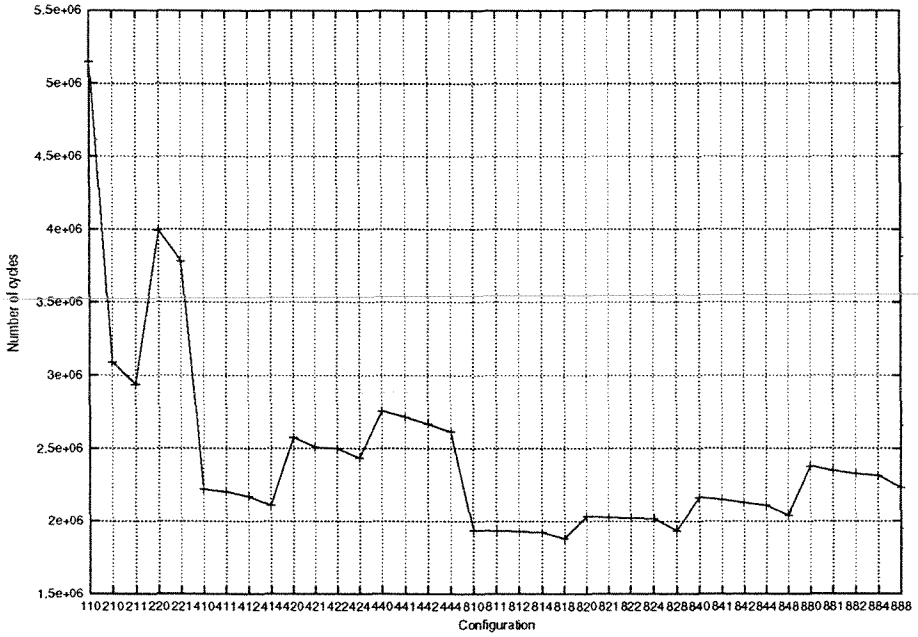


Figura 5.36: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *turb3d*.

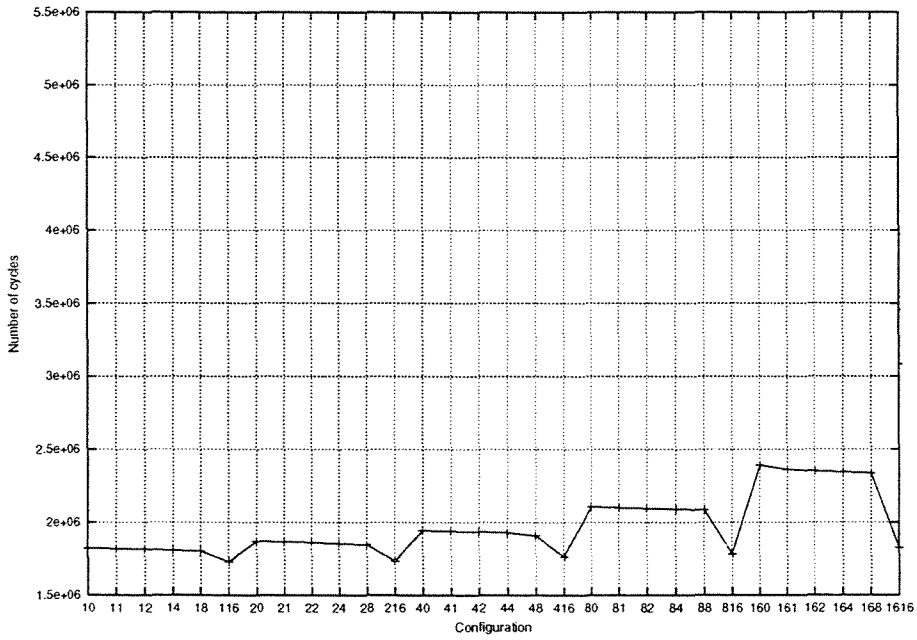
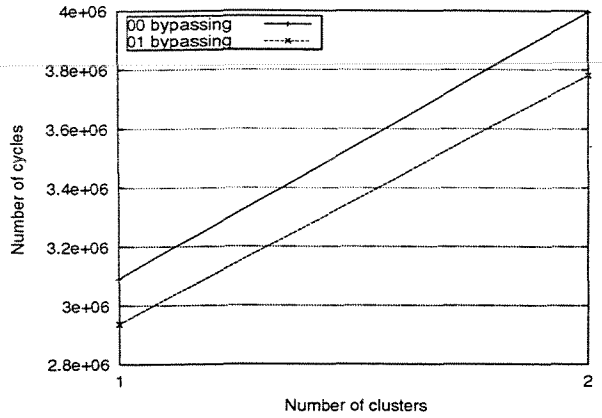
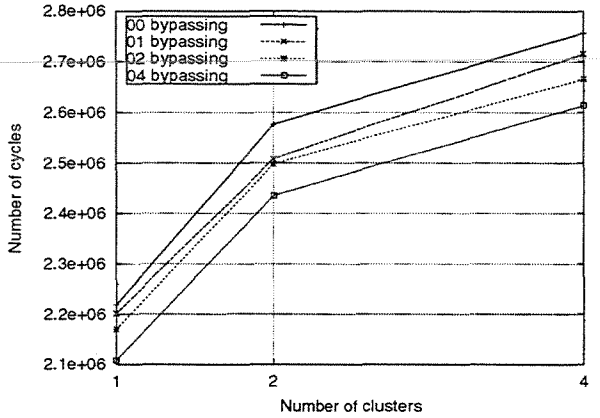


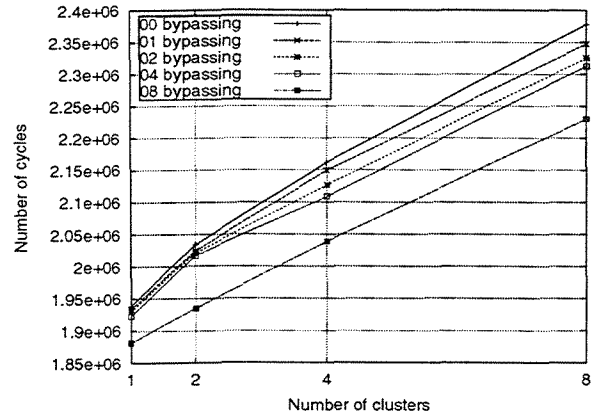
Figura 5.37: Número de ciclos gastos pelas configurações de 16 FUs para o programa *turb3d*.



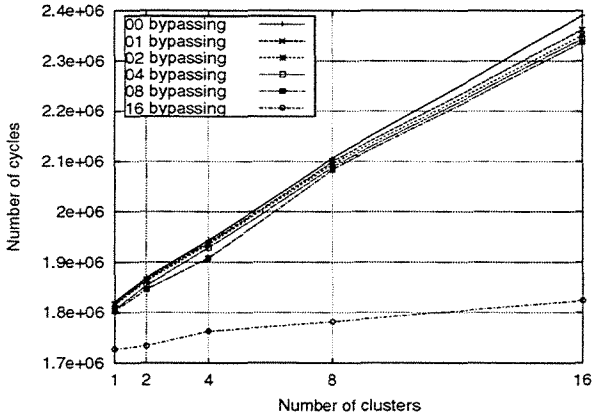
(a) 2 FUs



(b) 4 FUs

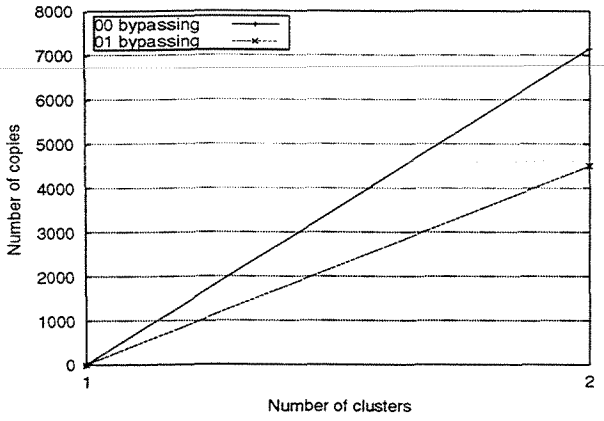


(c) 8 FUs

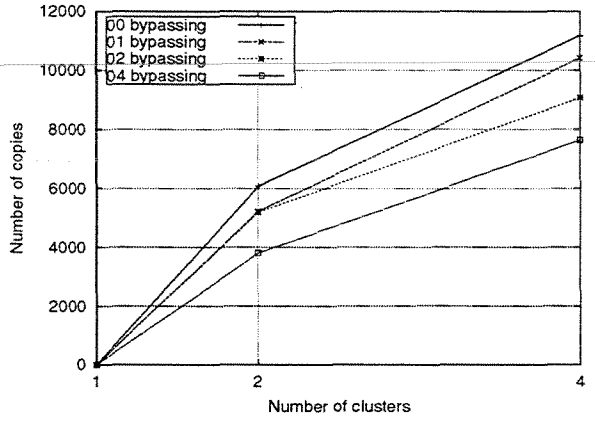


(d) 16 FUs

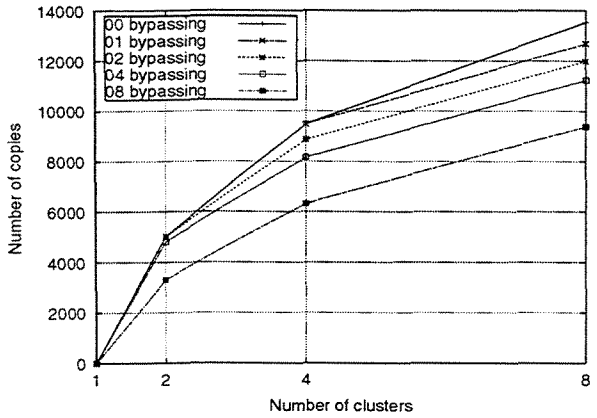
Figura 5.38: Resultados para o programa *turb3d* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



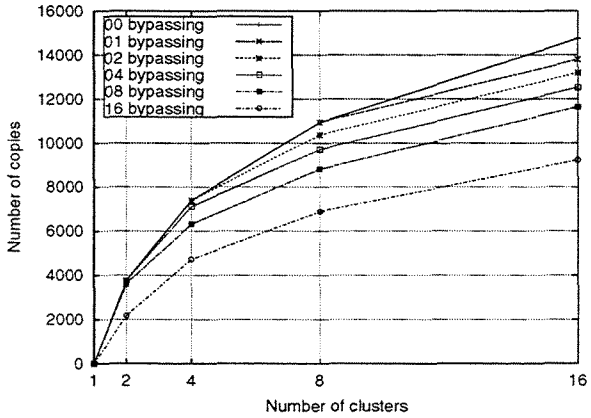
(a) 2 FUs



(b) 4 FUs



(c) 8 FUs



(d) 16 FUs

Figura 5.39: Número de instruções de cópia para o programa *turb3d* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

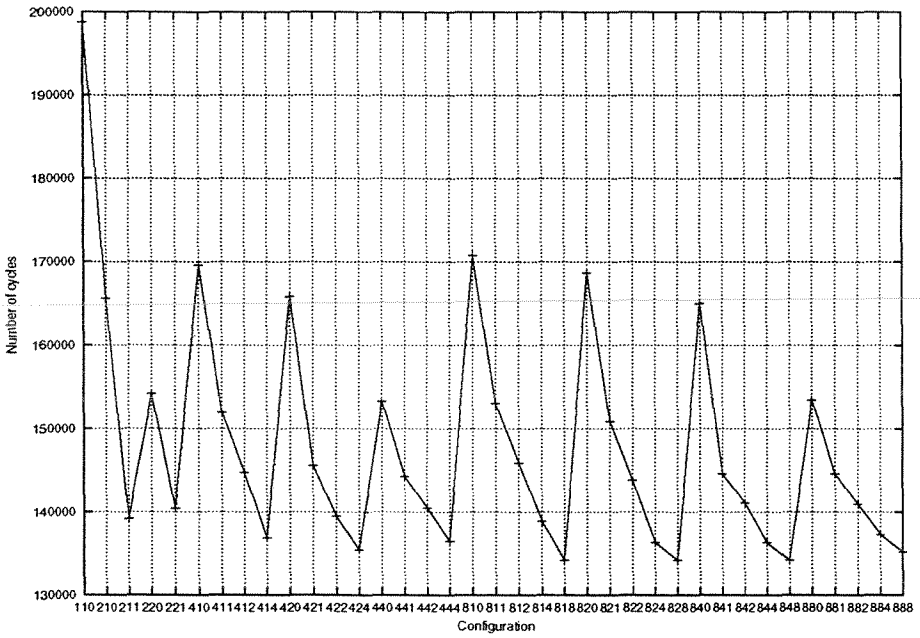


Figura 5.40: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *kalman*.

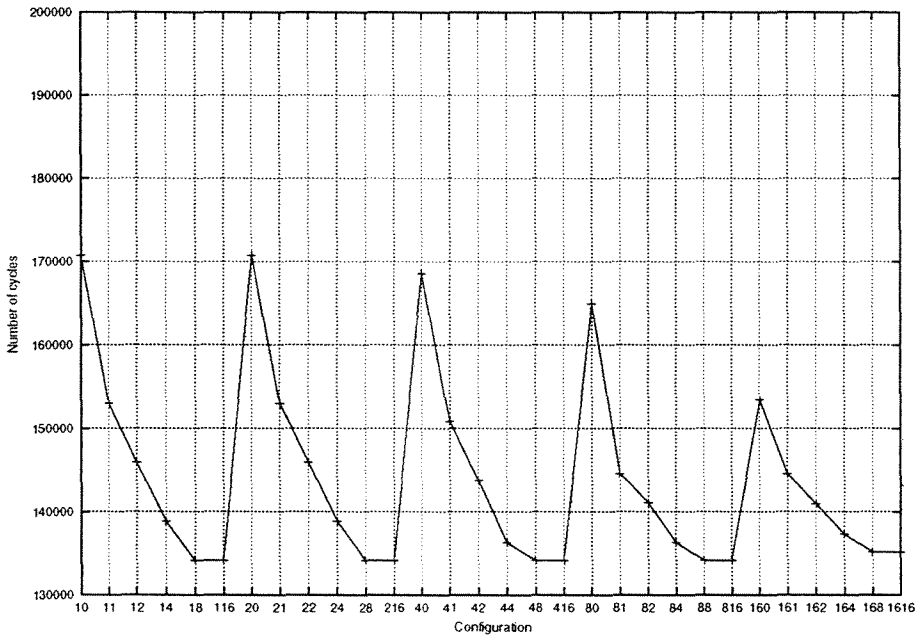
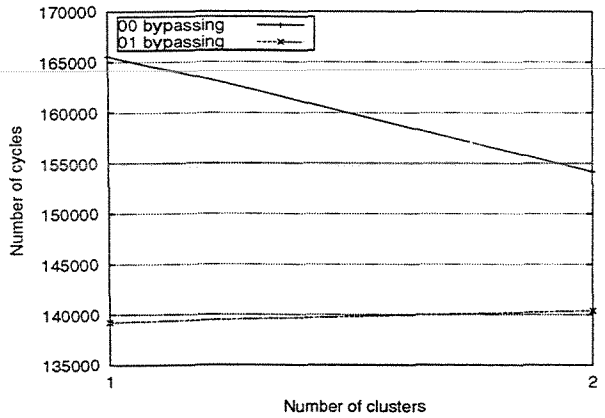
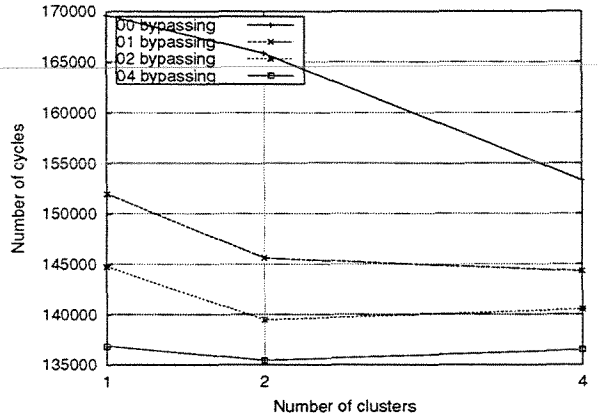


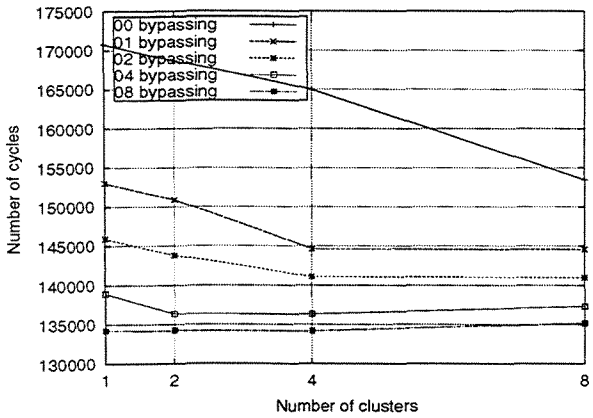
Figura 5.41: Número de ciclos gastos pelas configurações de 16 FUs para o programa *kalman*.



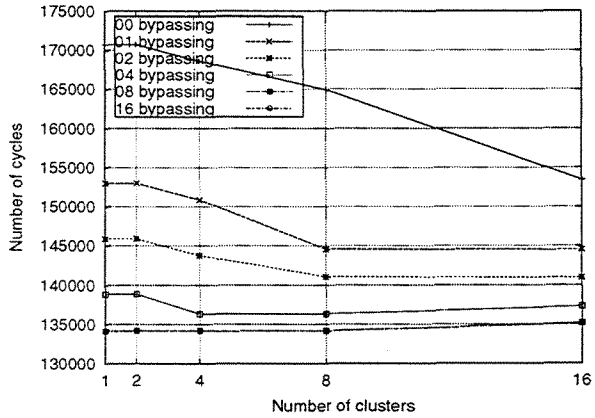
(a) 2 FUs



(b) 4 FUs

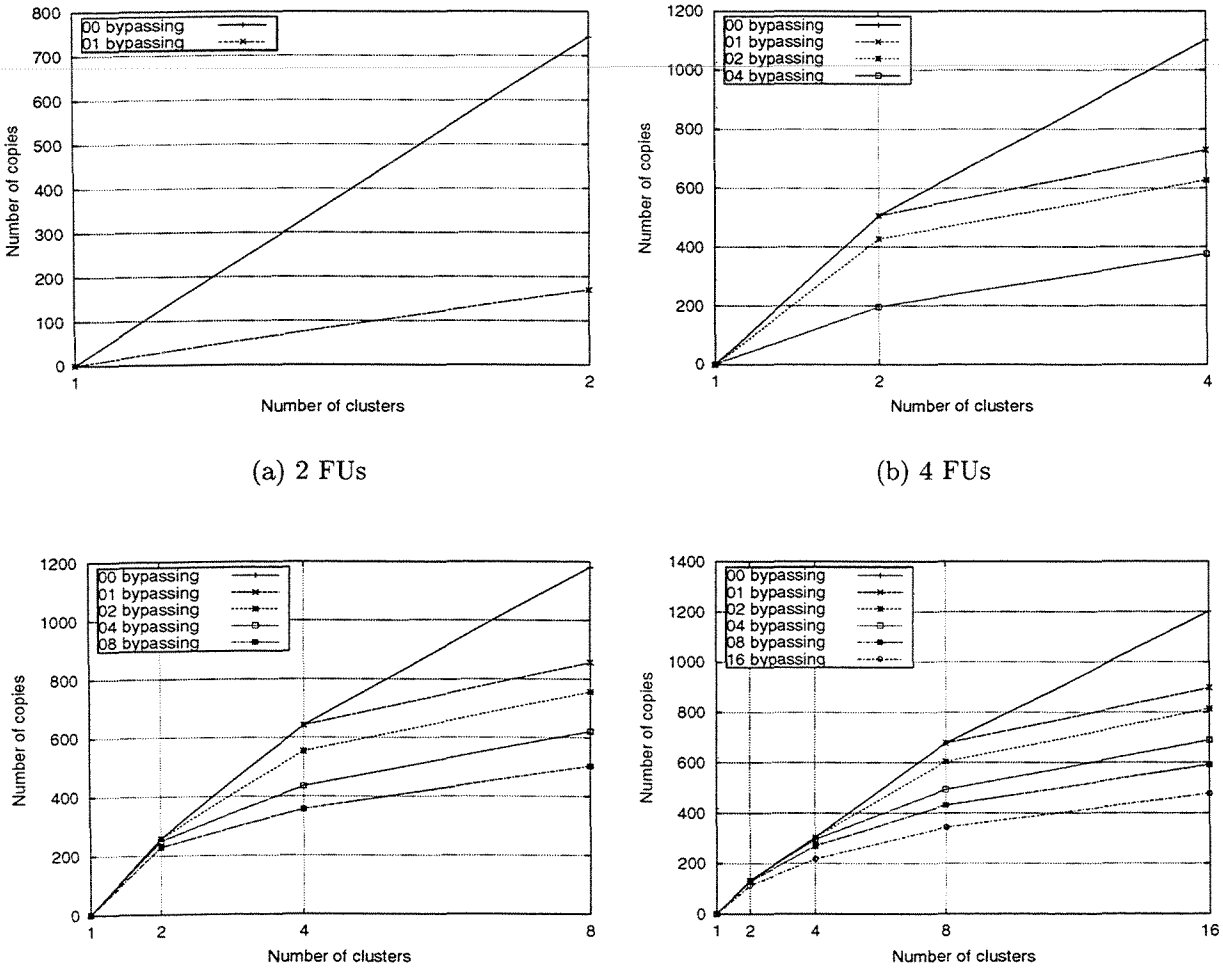


(c) 8 FUs



(d) 16 FUs

Figura 5.42: Resultados para o programa *kalman* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



(a) 2 FUs

(b) 4 FUs

(c) 8 FUs

(d) 16 FUs

Figura 5.43: Número de instruções de cópia para o programa *kalman* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

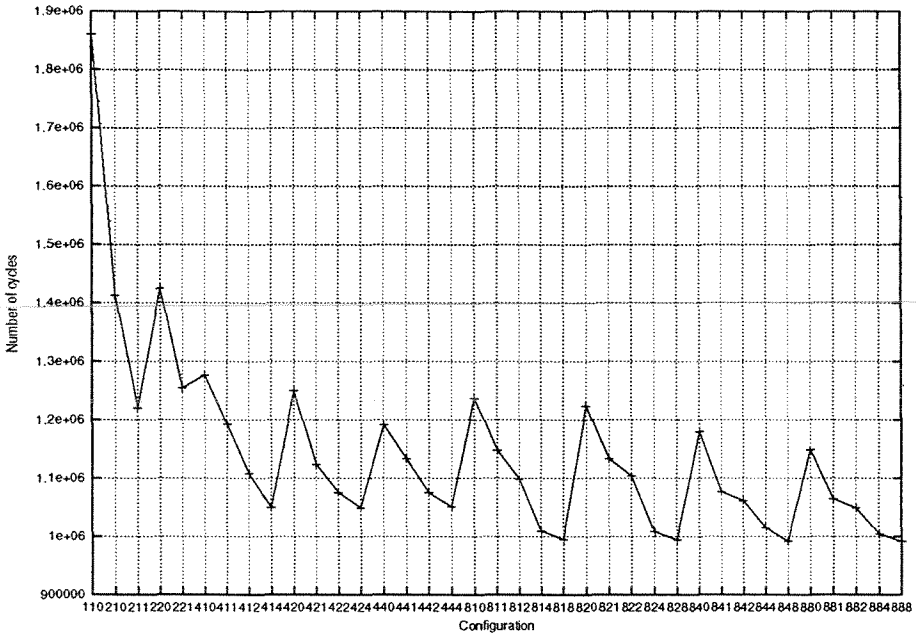


Figura 5.44: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *mpeg2enc*.

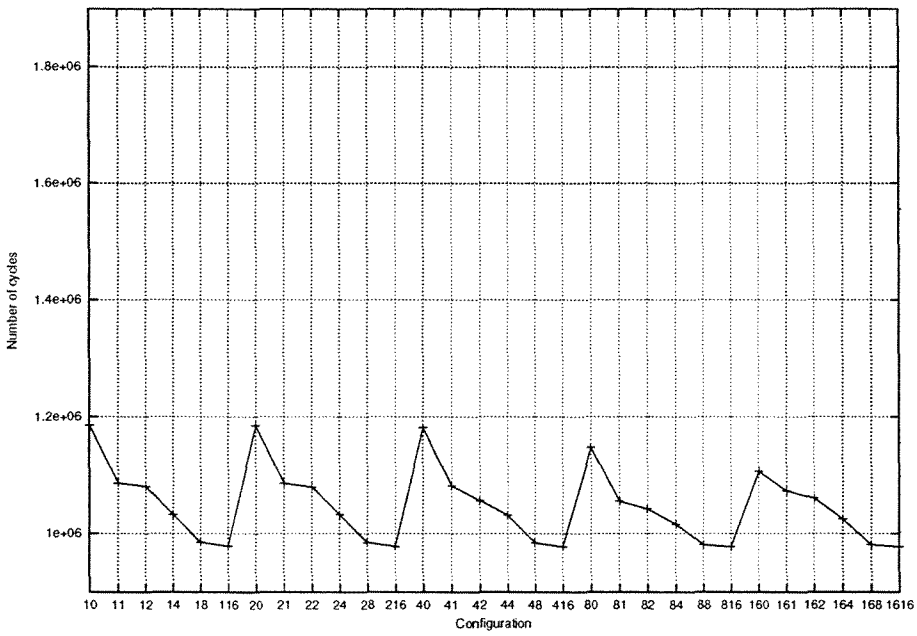
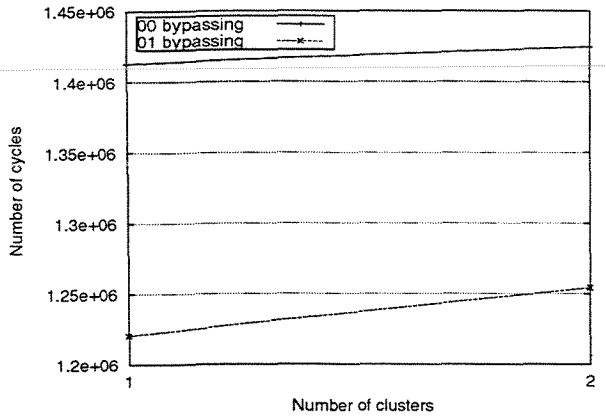
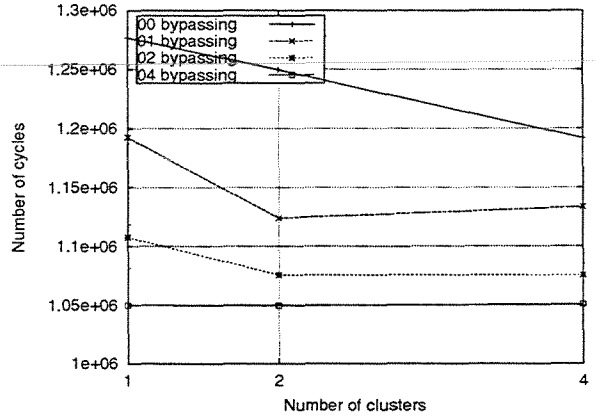


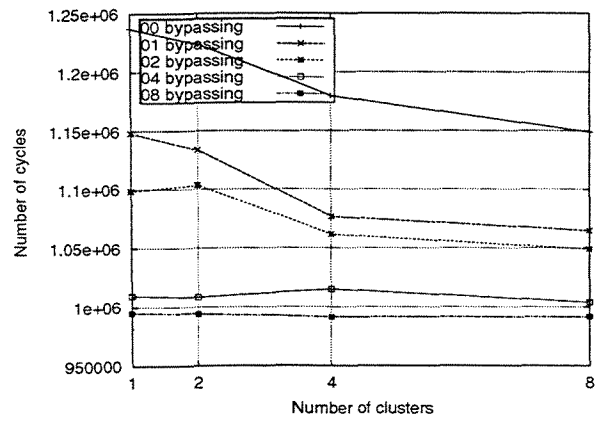
Figura 5.45: Número de ciclos gastos pelas configurações de 16 FUs para o programa *mpeg2enc*.



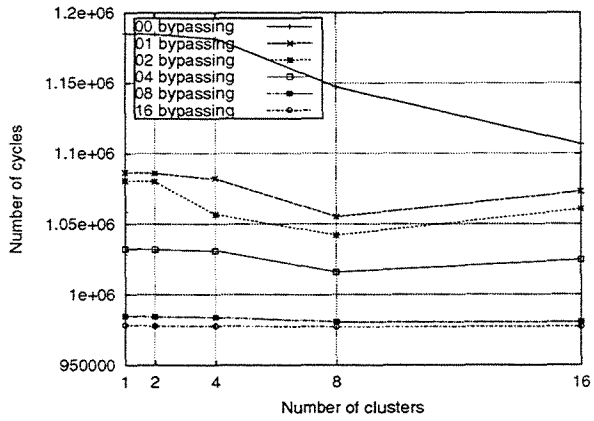
(a) 2 FUs



(b) 4 FUs

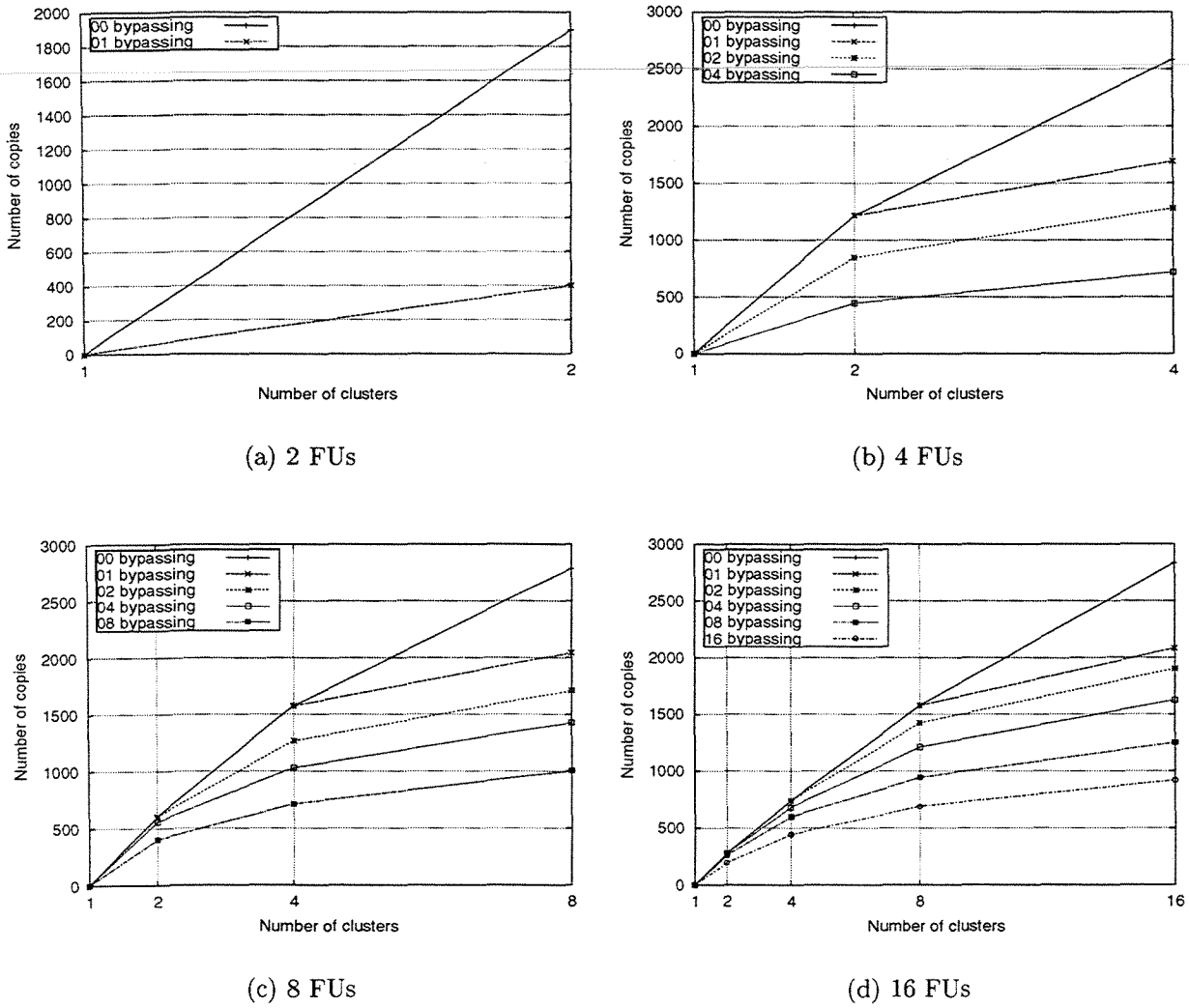


(c) 8 FUs



(d) 16 FUs

Figura 5.46: Resultados para o programa *mpeg2enc* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



(a) 2 FUs

(b) 4 FUs

(c) 8 FUs

(d) 16 FUs

Figura 5.47: Número de instruções de cópia para o programa *mpeg2enc* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

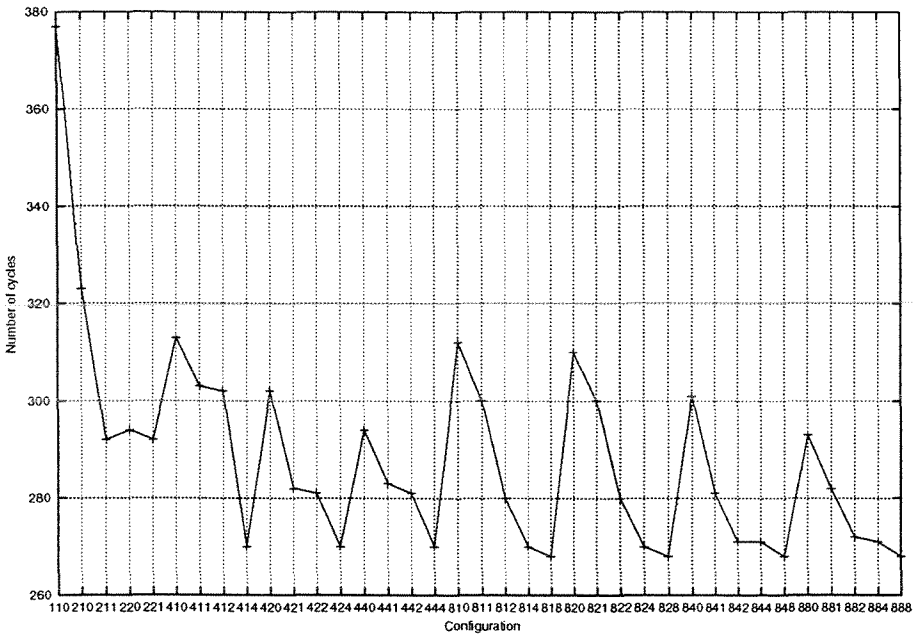


Figura 5.48: Número de ciclos gastos pelas configurações desde 1 FU até 8 FUs para o programa *eight*.

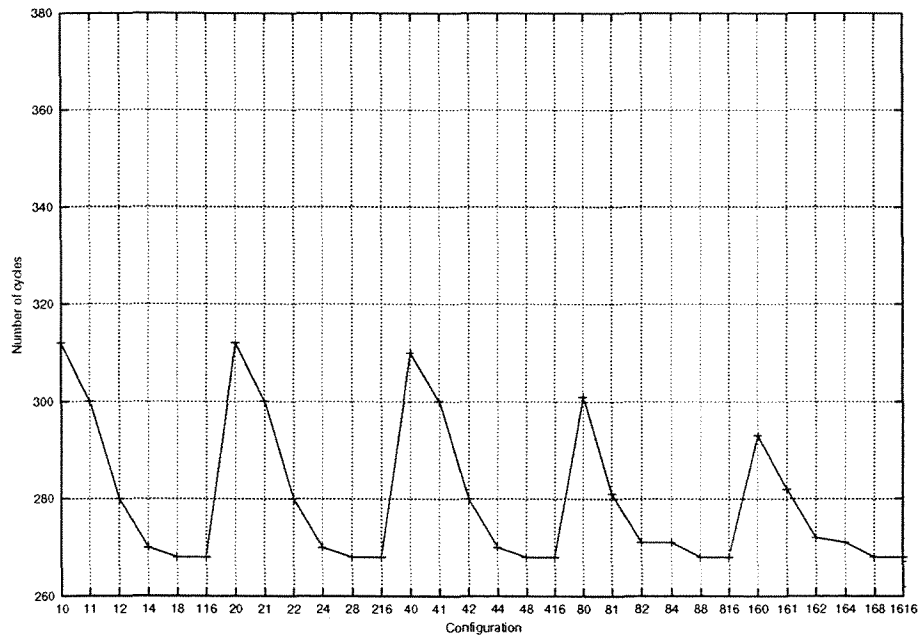
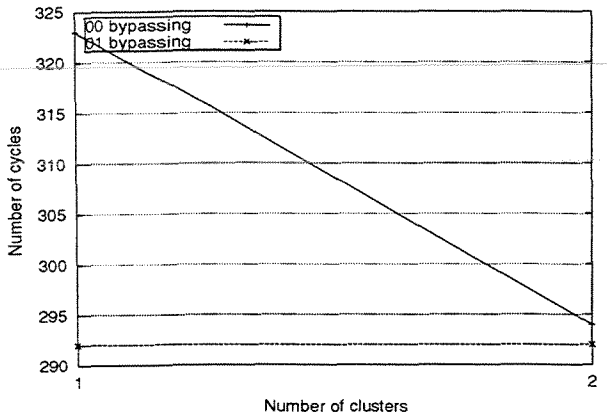
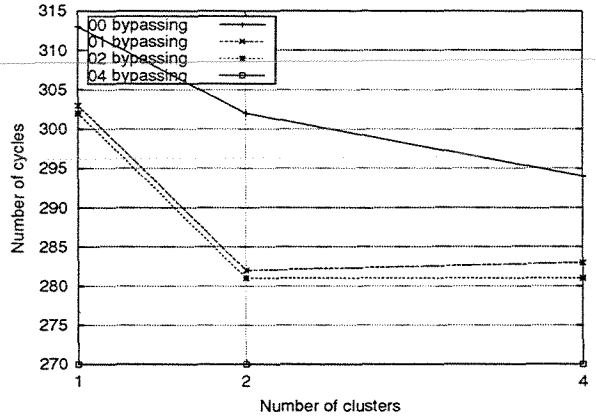


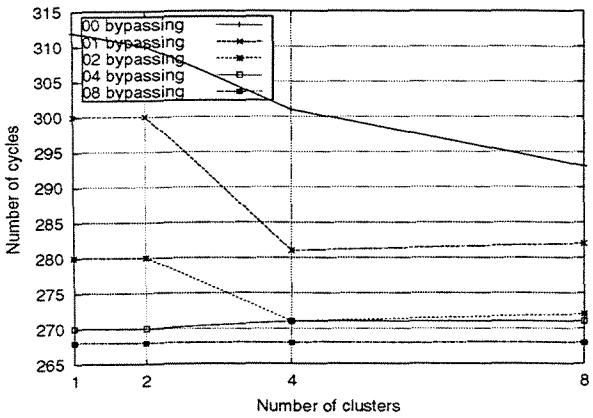
Figura 5.49: Número de ciclos gastos pelas configurações de 16 FUs para o programa *eight*.



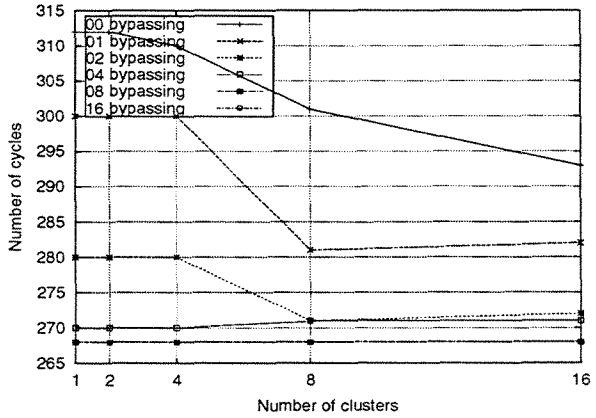
(a) 2 FUs



(b) 4 FUs

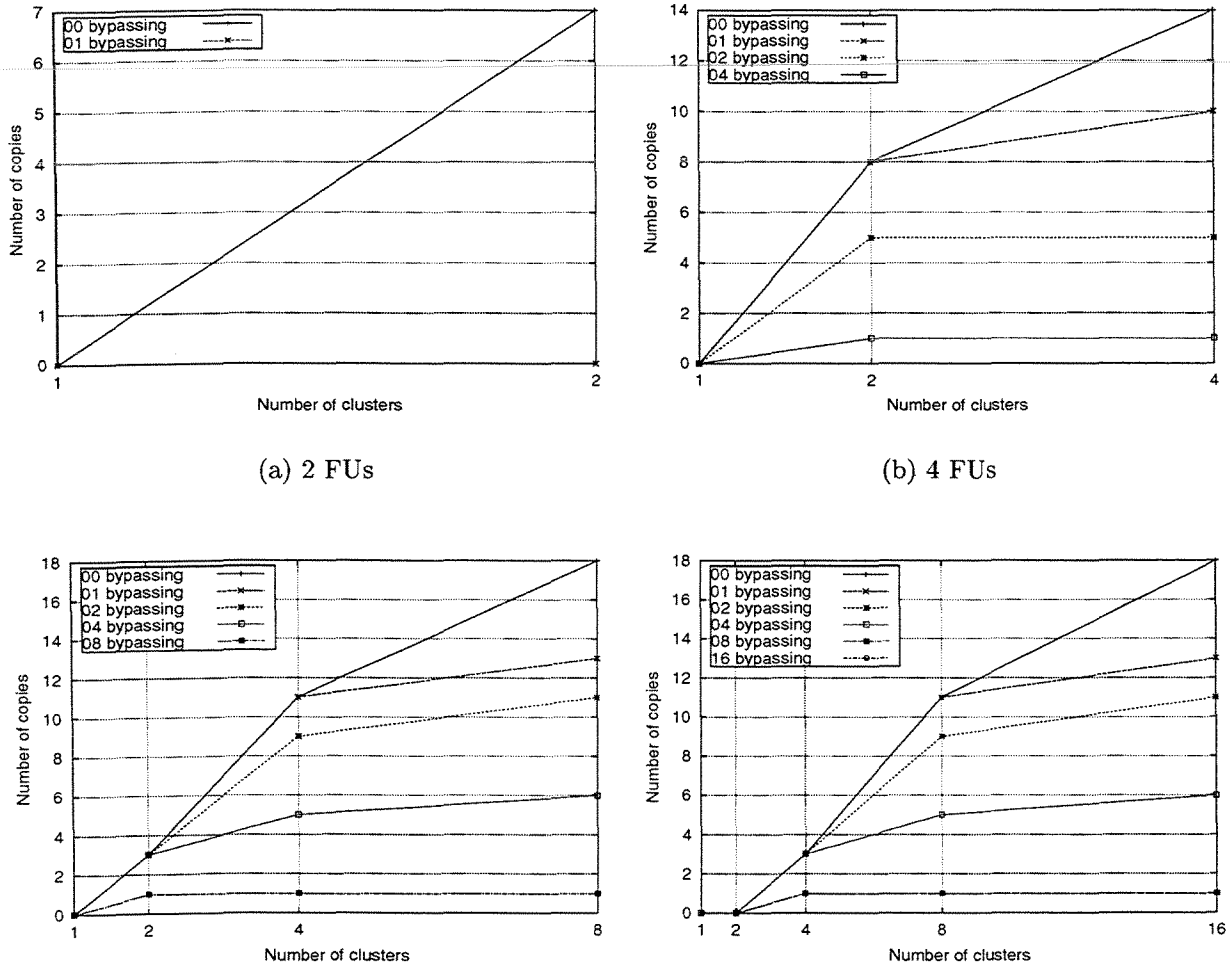


(c) 8 FUs



(d) 16 FUs

Figura 5.50: Resultados para o programa *eight* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.



(a) 2 FUs

(b) 4 FUs

(c) 8 FUs

(d) 16 FUs

Figura 5.51: Número de instruções de cópia para o programa *eight* considerando-se (a). 2, (b). 4, (c).8 e (d). 16 unidades funcionais.

5.9.3 Impacto do Particionamento e do *Bypassing* no *Cycle Time*

Conforme comentado em várias partes desta dissertação, o tamanho do ciclo de relógio pode ser afetado pelo aumento do número de portas dos bancos de registradores da arquitetura VLIW particionada. O tempo de acesso a um banco de registradores é aproximadamente proporcional ao logaritmo do número de portas de saída do mesmo [8]. É possível, portanto, que o caminho crítico do *pipeline* seja definido pela leitura dos operandos no banco de registradores, fazendo com que o ciclo de relógio seja diferente entre diferentes configurações de arquitetura. Assim, os valores mostrados na subseção 5.9.2 referem-se somente ao número de ciclos, mas não ao tempo de execução da aplicação.

Apresentamos no decorrer deste trabalho um *insight* sobre o efeito do particionamento da arquitetura no tamanho do ciclo de relógio. Uma análise precisa deste efeito está além do escopo desta dissertação, portanto apenas citamos como realizar uma estimativa da variação do tempo de acesso em relação à variação do número de portas de saída no banco de registradores.

Pode-se, entretanto, estimar alguns parâmetros referentes à variável "tempo", como por exemplo a relação entre os tempos de acesso de duas configurações diferentes "Cfg1" e "Cfg2". A expressão abaixo dá uma noção aproximada desta relação:

$$\frac{log(Output\ Ports\ Cfg1)}{log(Output\ Ports\ Cfg2)}$$

Desta maneira, uma configuração "410", por exemplo, possui um tempo de acesso ao banco de registradores 1.5 vezes maior do que uma configuração "420". Supor que esse seja o impacto no tamanho do ciclo de relógio torna-se um pouco mais complexo, pois vários fatores devem fazer parte dessa análise.

A quantidade de *bypass* adicionada à arquitetura também tem uma influência no *cycle time* da mesma. Entretanto, este impacto é muito difícil de ser estimado ou calculado, sendo um tópico que vai além das fronteiras desta dissertação.

Trabalhos futuros poderiam abordar profundamente os aspectos relacionados com os efeitos da especialização dos processadores proposta no tamanho do ciclo de relógio.

Capítulo 6

Conclusões e Trabalhos Futuros

Esta dissertação apresentou alguns aspectos relacionados com arquiteturas VLIW particionadas e propôs, além de algumas modificações nos modelos desta arquitetura encontrados na literatura, uma técnica de escalonamento de instruções que tem como objetivo a redução do número de ciclos de execução de aplicações específicas. O contexto de sistemas dedicados é inserido neste trabalho no momento em que a especialização do processador torna-se um dos pontos-chave do mesmo. Sistemas dedicados têm a característica de juntar, em um só grupo, projetistas de arquitetura e de compiladores. Quando a arquitetura sendo considerada é do tipo VLIW, esta amarração torna-se ainda mais forte. Desta forma, esta dissertação trata simultaneamente de assuntos relacionados com geração de código e especialização de processadores para aplicações específicas. Projetistas de arquiteturas podem valer-se dos resultados do algoritmo de escalonamento de instruções para adequarem o número de unidades funcionais, o número de bancos de registradores e o "volume" da rede de *bypassing* para um dado processador inserido em um sistema embarcado.

Um dos motivos prováveis pelo qual arquiteturas VLIW particionadas não tornaram-se (ainda) padrão em DSPs é o alto preço pago pelas instruções de cópia. Os DSPs atuais baseados no modelo VLIW particionado geralmente utilizam "truques" para mascarar este custo. Um exemplo prático são os *cross paths* X1 e X2 no processador TI C6201 [34]. Através deles, operandos de um dos dois *clusters* da arquitetura podem ser lidos por unidades funcionais pertencentes ao outro *cluster*.

Acreditamos que, pelo fato de estas arquiteturas não proverem uma rede de *bypassing* satisfatória, o número de cópias necessário é muito elevado. Além disso, como os operandos das instruções de cópia geralmente provêem unicamente do banco de registradores, estas instruções devem esperar que o registrador a ser copiado seja escrito no banco, aumentando ainda mais o custo das instruções de cópia quando a mesma está próxima da instrução que definiu o registrador.

A metodologia apresentada nesta dissertação sugere uma nova organização para os processadores VLIW particionados para aplicações específicas com o objetivo de tornar mais eficiente uso destes processadores. Pode-se notar pelos resultados apresentados no capítulo 5 que uma vasta exploração arquitetural pode ser realizada com as estratégias propostas.

Outra conclusão muito importante extraída deste trabalho diz respeito à transferência da complexidade do *hardware* para o compilador adotada na metodologia VLIW. Em parte essa estratégia é corretíssima, retirando por exemplo o escalonamento dinâmico dos processadores super-escalares e substituindo-o por um escalonamento estático via compilador. Entretanto, deve haver um balanceamento entre quais funcionalidades adicionar no *hardware* e quais tarefas deixar a cargo do compilador. Por exemplo, sem uma rede de *bypassing* satisfatória entre as unidades funcionais o compilador não conseguirá escalonar as operações dependentes de modo a sempre minimizar o número de ciclos. Certas vezes o compilador será obrigado a inserir NOPs para resolver *data hazards* devido à dependências de dados, degradando o desempenho da aplicação. Se permitirmos um certo nível de *hardware* extra, é possível que as tarefas do compilador sejam mantidas com a mesma complexidade porém o desempenho final seja melhorado. Sugere-se, portanto, que o conceito "transferir toda a complexidade para o compilador" adotado no modelo VLIW poderia ser levemente revisto. Como trabalhos futuros, pode-se citar alguns pontos-chave:

- Registradores de "contenção" podem ser inseridos nas linhas de *bypassing* no sentido de evitar um número ainda maior de instruções de cópia. Em outras palavras, duas unidades funcionais podem ter linhas de *bypassing* entre elas e mesmo assim necessitarem de cópias alguma vezes. Isso ocorre quando duas operações dependentes são escalonadas para serem executadas nestas unidades e a distância entre as operações é elevada. Com os registradores de contenção o dado seria temporariamente armazenado e logo após lido pela operação que o utiliza.

- O número de estágios do *pipeline* da arquitetura pode ser revisto. Um *pipeline* com 4 estágios apenas pode reduzir o número de NOPs de dois para um na tabela 2.2. Neste novo *pipeline* o único tipo de endereçamento à memória permitido é modo indireto por registrador, no qual o cálculo do endereço a ser acessado é calculado numa instrução de *add* anterior às instruções de *load* ou *store*.

- Considerar arquiteturas heterogêneas ao invés de uma arquitetura onde qualquer unidade funcional pode executar qualquer tipo de operação.

- Explorar novas heurísticas para os algoritmos de inserção de cópias, inserção de NOPs e para o escalonamento de instruções propriamente dito.

Bibliografia

[1] IEEE Micro Vol. 20(5). http://www.computer.org/micro/mi2000/m5toc.htm, Sep/Oct. 2000. Papers about Itanium Microprocessor.

[2] Arthur Abnous and Nader Bagherzadeh. Pipelining and bypassing in a VLIW processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):658–663, June 1994.

[3] Arthur Abnous and Nader Bagherzadeh. Architectural design and analysis of a VLIW processor. *International Journal of Computers and Electrical Engineering*, 21(2):119–142, 1995.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[5] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the Euro. Symp. on Programming, Springer-Verlag, Berlin*, pages 308–317, 1988.

[6] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO)*, pages 72 - 80, 1992.

[7] Guido Araujo. *Code Generation Algorithms for DSP Processors.* PhD thesis, Princeton University, 1997.

[8] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Design considerations for limited connectivity VLIW architectures. Technical Report Technical Report 92-59, UC Irvine, ICS Dept., 1992.

[9] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Multi ported register file complexity analysis. Technical Report Technical Report 92-58, UC Irvine, ICS Dept., 1992.

[10] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO)*, pages 292 – 300, 1992.

[11] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Architectural tradeoff analysis of partitioned VLIWs. Technical Report Technical Report 94-14, UC Irvine, ICS Dept., 1994.

[12] E.G. Coffman. *Computer and Job-shop Scheduling Theory.* John Wiley and Sons, New York, New York, 1976.

[13] Robert P. Colwell, Robert P. Nix, John J. O'Donnel, David B. Papwoth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967, August 1988.

[14] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA.* John Wiley and Sons, England, 1997.

[15] S. Davidson, D. Landskov, B.D. Shriver, and P.W. Mallet. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, 30(7):460–477, 1981.

[16] K. Ebcioglu. Some design ideas for a VLIW architecture for sequential natured software. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, 1988.

[17] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1986.

[18] P. Faraboshchi, G. Desoli, and J. A. Fisher. Clustered instruction-level parallel processors. Technical Report Technical Report HPL-98-204, HP Labs, USA, 1998.

[19] Edil S. T. Fernandes. Personal communication, 2001.

[20] Marcio Merino Fernandes, Josep Llosa, and Nigel Topham. Partitioned schedules for clustered VLIW architectures. In *IEEE/ACM International Parallel Processing Symposium*, 1998.

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319 – 349, July 1987.

[22] Joseph A. Fisher. Trace scheduling: A technique for global microcode compactation. *IEEE Transactions on Computers*, c-30(1):478 – 490, July 1981.

[23] Joseph A. Fisher. Very long instruction word architecture and the ELI-512. In *Proceedings of the 10th Symposium on Computer Architectures*, pages 140–150, June 1983.

[24] Joseph A. Fisher. The VLIW machine: A multiprocessor for compiling scientific code. In *Proceedings of the IEEE*, pages 45 – 53, July 1984.

[25] J. Fridman and Zvi Greefield. The tigersharc DSP architecture. In *IEEE Micro*, pages 66–76, Jan-Feb 2000.

[26] Daniel G. Gajski and Nikil Dutt. *High-Level Synthesis.* Morgan Kaufmman, Irvine, CA, 1991.

[27] M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[28] Franco Gasperoni. Compilation techniques for VLIW architectures. Technical Report 66741, IBM Research Division, T.J. Watson Research Center, NY, September 1989.

[29] GNU GCC. http://www.gnu.org.

[30] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421 – 431, April 1990.

[31] J. A. Henessy and David L. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann Publishers, 1990.

[32] W. W. Hwu et al. IMPACT advanced compiler technology. http://www.crhc.uiuc.edu/IMPACT/index.html.

[33] W. W. Hwu et al. The superblock: An efficient technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7, May 1993.

[34] Texas Instruments. *TMS320C62xx CPU and Instruction Set Reference Guide*, 1998. www.ti.com/sc/c6x.

[35] Margarida F. Jacome, Gustavo de Veciana, and Viktor Lapinskii. Exploring performance tradeoffs for clustered VLIW ASIPs. In *International Conference on Computer-Aided Design*, 2000.

[36] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and super-pipelined machines. In *Third International Symposium on*

Architectural Support for Programming Languages and Operating Systems, pages 272–282, 1989.

[37] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[38] Monica S. Lam. Software pipelining: An efficient scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 318 – 328, 1988.

[39] James Larus. EEL Executable Editing Library. http://www.cs.wisc.edu/ larus/eel.html.

[40] C.H. Lee, C.I. Park, and M. Kim. Efficient algorithm for graph partitioning problem using a problem transformation method. *Computer Aided Design*, 21(10):611, December 1989.

[41] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121 – 141, July 1979.

[42] Rainer Leupers. Instruction scheduling for clustered VLIW DSPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2000.

[43] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO)*, 1992.

[44] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[45] Alexandru Nicolau. A fine-grain parallelizing compiler. Technical Report TR-86-792, Dept. of Comp. Sci., Cornell University, Ithaca, NY, Dec. 1986.

[46] Emre Ozer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31th International Symposium on Microarchitecture (MICRO)*, 1998.

[47] Emre Ozer and Thomas M. Conte. Optimal cluster scheduling for a VLIW machine. Technical report, Dept. of Elec. and Comp. Eng., North Carolina State University, 1998.

[48] Emre Ozer and Thomas M. Conte. Unified cluster assignment and instruction scheduling for clustered VLIW microarchitectures. Technical report, Dept. of Elec. and Comp. Eng., North Carolina State University, 1998.

[49] J. C. H. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Labs, Palo Alto, CA, May 1991.

[50] Philips. Trimedia processors. http://www-us.semiconductors.philips.com/trimedia/.

[51] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The cydra5 departamental supercomputer. *IEEE Computer*, 22:12–35, January 1989.

[52] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO)*, pages 63–74, 1994.

[53] Microprocessor Report. Map1000 unfolds at equator. Technical report, Equator, 1998.

[54] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *26th International Symposium on High-Performance Computer Architecture*, May 1999.

[55] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. Power exploration for embedded VLIW architectures. In *ICCAD Conference*, 2000.

[56] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon. Exploting data forwarding to reduce the power budget of VLIW embedded processors. In *DATE Conference*, 2001.

[57] Jesus Sanchez and Antonio Gonzalez. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *International Conference on Parallel Processing (ICPP)*, 2000.

[58] Jesus Sanchez and Antonio Gonzalez. Instruction scheduling for clustered VLIW architectures. In *International Symposium on System Synthesis (ISSS)*, 2000.

[59] Jesus Sanchez and Antonio Gonzalez. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proceedings of the 33th International Symposium on Microarchitecture (MICRO)*, 2000.

[60] Luiz C. V. Santos. *Exploting Instruction-Level Parallelism : A Construtive Aprroach.* PhD thesis, University of Technology, The Netherlands, November 1998.

[61] Siemens. www.siemens.de/ic/products/cd/english/index, 1999.

[62] Will Strauss. The 1999 DSP market heats up. www.forwardconcepts.com, 1999.

[63] R.M. Tomasolo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):232–233, Jan. 1967.

[64] David W. Wall. Limits of instruction-level parallelism. In *Digital Equipment Corporation Technical Report*, pages 176–188, 1991.

[65] R. Yung and N.C. Wilhelm. Caching processor general registers. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 307–312, 1995.