DIONE JONATHAN FERRARI

APLICAÇÃO DE LOOP PIPELINING E LOOP UNROLLING À SÍNTESE DE ALTO NÍVEL

FLORIANÓPOLIS – SC
2002

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DIONE JONATHAN FERRARI

APLICAÇÃO DE LOOP PIPELINING E LOOP UNROLLING À SÍNTESE DE ALTO NÍVEL

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador:

LUIZ CLÁUDIO VILLAR DOS SANTOS

Florianópolis, Julho de 2002.

APLICAÇÃO DE LOOP PIPELINING E LOOP UNROLLING À SÍNTESE DE ALTO NÍVEL

Dione Jonathan Ferrari

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação (Sistema de Computação) e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Fernando Alvaro Ostuni Gauthier (coordenador)

Banca Examinadora

Luiz Cláudio Villar dos Santos (orientador)

Ricardo Augusto da Luz Reis

Olinto José Varela Furtado

Primeiramente agradeço a Deus pela força, saúde e tudo o mais que Ele me proporcionou.

Em segundo, agradeço aos meus familiares que durante todo o tempo, mesmo estando distantes, se mostraram presentes, me apoiando e me incentivando em mais essa etapa de minha vida.

Agradeço também ao professor e amigo Luiz Cláudio Villar dos Santos pela oportunidade de pesquisa nesse grupo e principalmente pela orientação neste trabalho.

Também quero agradecer a todos os membros da equipe do projeto OASIS: Modelagem, Síntese e Otimização de Arquiteturas para Sistemas Digitais, em especial ao bolsista de iniciação científica Felipe Vieira Klein pelo auxílio na elaboração do programa-protótipo que ampara este trabalho. Agradeço também pelo companheirismo e amizade que me foram proporcionados dentro e fora do LAPS.

APLICAÇÃO DE LOOP PIPELINING E LOOP UNROLLING À SÍNTESE DE ALTO NÍVEL

Dione Jonathan Ferrari

Julho / 2002

Orientador: Luiz Cláudio Villar dos Santos.

Área de Conhecimento: Sistemas de Computação.

Palavras-chave: Síntese de Alto Nível, Escalonamento, Loop Pipelining, Loop Unrolling.

Número de Páginas: 58.

RESUMO:

Este trabalho tem como objetivo resolver um problema clássico da Síntese de Alto Nível através de uma abordagem orientada à exploração de soluções alternativas.

O problema consiste no *escalonamento* de operações de um dado algoritmo sob *restrição de recursos* físicos de forma que cada operação é executada respeitando a ordem de precedência imposta pelo algoritmo.

Para abordar o problema acima, utilizou-se as técnicas de *Loop Pipelining* e *Loop Unrolling*, onde operações de diferentes iterações podem ser executadas em um mesmo estado. Estas técnicas, por exporem mais paralelismo, permitem uma melhor utilização dos recursos.

Este trabalho descreve a abordagem proposta, a modelagem que a ampara e a implementação de ferramentas que a suportam (escalonador e paralelizador). São apresentados resultados experimentais obtidos a partir de exemplos clássicos da literatura.

LOOP PIPELINING AND LOOP UNROLLING APPLICATION ON HIGH LEVEL SYNTHESIS

Dione Jonathan Ferrari

July / 2002

Advisor: Luiz Cláudio Villar dos Santos.

Area of Concentration: Computing Systems.

Keywords: High level synthesis, Scheduling, Loop Pipelining, Loop Unrolling.

Number of Pages: 58.

ABSTRACT:

This work has as objective to solve a classical High Level Synthesis problem through an approach oriented to the exploration of alternative solutions.

The problem consists of scheduling operations of a given algorithm under physical resource constraints, in such away that each operation is executed, respecting the precedence order imposed by the algorithm.

To approach the problem above, the Loop Pipelining and Loop Unrolling techniques were used. This technique allows that operations from different iterations can be executed in a same state. These techniques, for exposing more parallelism, allows a better use of the resources usage.

This work describes the proposed approach, its underlying modeling and the implementation of its supporting tools (scheduler and parallelizer). Classical examples from the literature are used to generate experimented results.

Sumário

Lista de Figuras

Lista de Gráficos

Lista de Tabelas

Lista de Algoritmos

Capítulo

1 Introdução

A evolução dos sistemas computacionais está tornando cada vez mais comum o uso de *Sistemas Computacionais Embutidos* em equipamentos que fazem parte de nosso cotidiano (telefone celular, secretária eletrônica, eletrodomésticos em geral, etc.). Junto com essa evolução, há a competição de indústrias no mercado de sistemas**,** que requer um projeto rápido para que as coloque na liderança desse mercado.

Por esses e outros motivos surgiram ferramentas para o projeto de sistemas. Essas ferramentas são programas que auxiliam o desenvolvimento de projetos, através de técnicas de CAD ("Computer-Aided Design"). Por permitirem a automação do projeto eletrônico, também são conhecidas como EDA ("Electronic Design Automation").

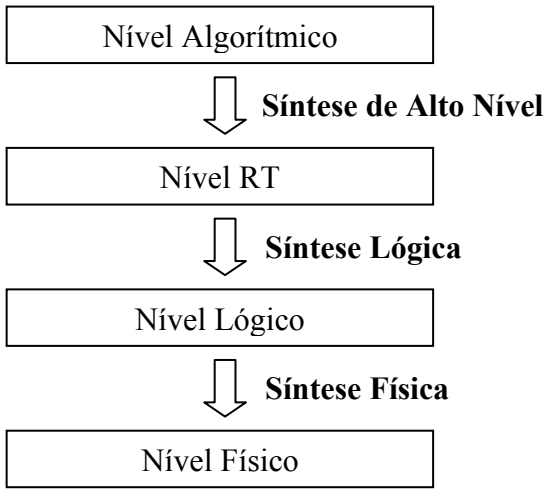


Figura 1.1: Os níveis de abstração do projeto de sistemas digitais.

Muitos dos Sistemas Embutidos fazem uso de circuitos integrados de aplicação específica ou ASICs ("Application Specific Integrated Circuit"). As principais etapas do projeto de um ASIC são: a *síntese*, que consiste em se obter um circuito que implemente a especificação do mesmo; a *validação*, que verifica se o circuito sintetizado funcionará; e o *teste* que garante que todas as funções verificadas são executadas corretamente.

O projeto de sistemas é usualmente visto como uma sucessão de etapas que envolvem diferentes níveis de abstração, conforme ilustra a Figura 1.1. O primeiro nível, chamado de nível algorítmico, descreve a função de um sistema na forma de um algoritmo sem se preocupar com a forma em que o sistema será implementado. O nível de transferência entre registradores ou RT ("register-transfer") descreve a estrutura do sistema como um circuito composto de unidades funcionais (somadores, multiplicadores, subtratores, etc.), elementos de memória (registradores, bancos de memória, etc.) e elementos de interconexão (barramentos, etc.). Em seguida, vem o nível lógico que descreve o sistema em termos mais básicos como portas lógicas e flip-flops. O nível físico corresponde a uma descrição do sistema como a interconexão de transistores, resistores e capacitores.

A Síntese de Alto Nível, objeto deste estudo, transforma uma especificação do comportamento de um sistema no nível algorítmico para uma arquitetura no nível RT que implementa aquela especificação [SANT00] e pode ser vista em quatro etapas:

- A *seleção de módulos*, que determina os tipos de módulos que serão necessários na unidade operativa;
- A *alocação*, responsável por determinar quantos módulos de cada tipo serão necessários;
- O *escalonamento*, que define quando as operações são executadas;
- A *ligação*, que define em qual módulo específico cada operação escalonada será executada.

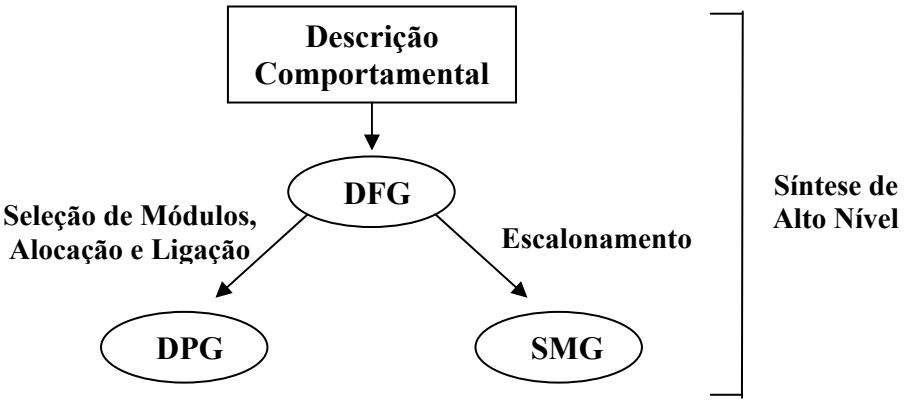


Figura 1.2: Uma visão geral dos modelos utilizados na Síntese.

Para se automatizar esse processo é preciso que modelos sejam definidos. Os modelos utilizados neste trabalho estão representados na Figura 1.2. O grafo de fluxo de dados ou DFG ("data-flow graph") descreve o comportamento de um algoritmo na forma de um grafo, incorporando suas operações e suas dependências. A partir dele, dois grafos são obtidos. Um dos grafos, denominado DPG ("datapath graph"), modela a estrutura da unidade operativa (nível RT). O outro grafo, denominado SMG ("state machine graph"), modela a unidade de controle. Em um SMG, cada vértice corresponde a um estado e as arestas representam as transições entre os estados.

A seguir é mostrado um exemplo simplificado da Síntese de Alto Nível. A Figura 1.3a descreve o comportamento de um sistema e seu respectivo DFG pode ser visualizado na Figura 1.3b.

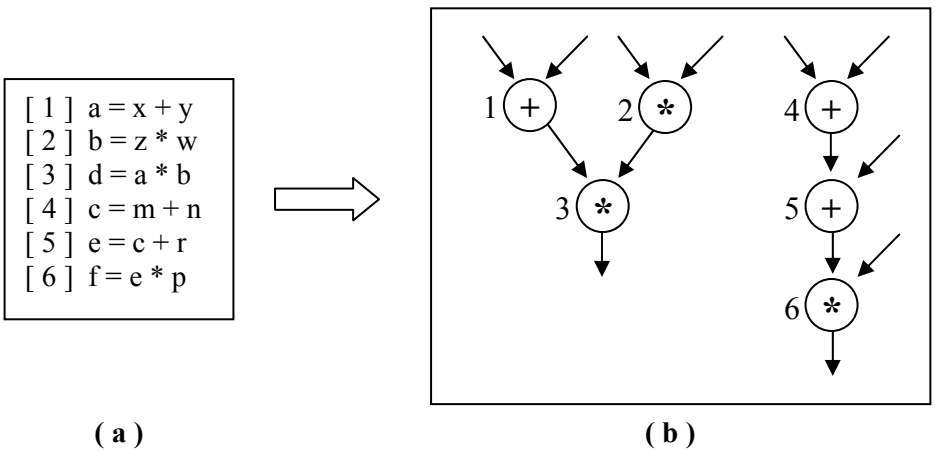


Figura 1.3: Esboço de uma descrição comportamental e seu respectivo DFG.

A síntese desse sistema irá resultar nos dois grafos mostrados na Figura 1.4. No DPG, esboçado de forma meramente esquemática, há três tipos de módulos: unidades funcionais (somador e multiplicador), conjunto de registradores e barramentos. Os números associados às unidades funcionais mostram quais operações do DFG são executadas nesse módulo. Note que, neste exemplo, as operações "1", "4" e "5" são associadas ao somador; as operações do "2", "3" e "6", ao multiplicador.

Os números associados aos vértices do SMG mostram em qual estado as operações são executadas. Por exemplo, as operações "1" e "2" podem ser executadas simultaneamente no primeiro estado, pois são independentes e ocupam módulos diferentes. Note que a operação "3" não pode ser executada no primeiro estado por dois motivos: esta operação requer um somador, ocupado pela operação "2" naquele estado (*restrição de recursos*); os valores consumidos por "3" são produzidos por "1" e "2" no primeiro estado (*restrição de precedência*).

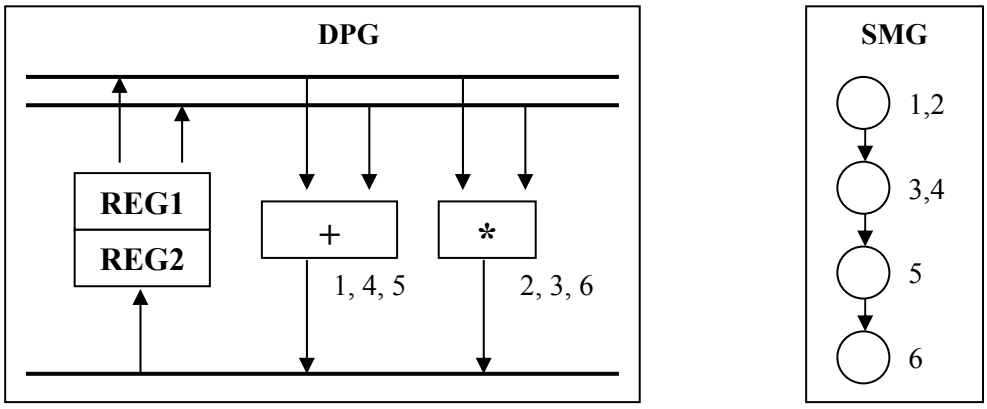


Figura 1.4: DPG e SMG sintetizados.

O objetivo deste trabalho é resolver um problema clássico da Síntese de Alto Nível**,** através de uma abordagem orientada à exploração automática de soluções alternativas. O problema consiste no escalonamento das operações, com restrição de recursos, utilizando a técnica denominada *paralelização de laços*, onde operações de diferentes iterações do laço podem ser executadas em um mesmo estado. A aplicação dessas técnicas permite explorar melhor o paralelismo contido na descrição comportamental, levando a soluções de melhor qualidade.

Esta dissertação está organizada da seguinte forma: O Capítulo 2 trata da modelagem do problema a ser resolvido e termina com uma breve revisão bibliográfica das técnicas clássicas de resolução. O Capítulo 3 apresenta a abordagem adotada. O Capítulo 4 relata a implementação da abordagem, os experimentos realizados e os resultados obtidos. Nesse capítulo, os resultados obtidos são analisados e comparados com os resultados

encontrados na literatura. As conclusões e as perspectivas de continuidade da pesquisa em trabalhos futuros são discutidas no Capítulo 5.

Além disso, os Anexos A e B detalham, para fins ilustrativos, duas amostras de resultados experimentais: um obtido via “loop pipelining”, outro obtido via “loop unrolling”. Ademais, o anexo C discute diferenças nas métricas de latência.

As principais contribuições desta dissertação estão listadas abaixo e serão detalhadas no Capítulo 5:

- Escolha, adaptação e implementação de um algoritmo de paralelização de laços;
- Análise do impacto das técnicas de paralelização.

Capítulo

2 Modelagem e Formulação do Problema

2.1 Definições Básicas

A Figura 2.1 mostra um exemplo de escalonamento utilizando "loop pipelining". Este exemplo servirá para dar uma noção intuitiva dos modelos a serem definidos formalmente mais tarde.

Assuma que o DFG ilustrado na Figura 2.1a representa o corpo de um laço e suponha que as operações são re-executadas em cada iteração desse laço. Assuma também que um somador e um multiplicador estão disponíveis. Por fim, suponha que a execução de uma operação leva um único ciclo em ambos os operadores, ou seja, cada operação pode ser executada em único estado.

Dada a operação "a", o símbolo a^i representa a instância da operação "a" na i-ésima iteração do laço. Por simplicidade, supõe-se que o laço execute um número infinito de vezes. Por isso, não se representa o teste para controlar a saída do laço.

A Figura 2.1b mostra um escalonamento sem utilizar a paralelização das operações do laço. Neste exemplo, todas as instâncias das operações pertencentes a uma dada iteração são executadas para só então se iniciar a execução de instâncias pertencentes à próxima iteração. Cada iteração do laço leva 4 ciclos para executar. Note que no estado s_2 o multiplicador não estava sendo usado e no estado s_3 acontece o mesmo com o somador. Uma alternativa seria permitir que a próxima iteração do laço pudesse começar antes da iteração anterior terminar. Esta técnica é conhecida como paralelização de laços. Por exemplo, a execução da operação "b" na segunda iteração pode ser feita simultaneamente com a execução da operação "e" na primeira iteração, como mostra a

Figura 2.1c. Como conseqüência, ao invés de cada iteração levar 4 ciclos para completar, novos resultados são produzidos depois de cada 3 ciclos. Diz-se que o corpo do laço está executando em *paralelo*, pois instâncias pertencentes a diferentes iterações executam simultaneamente (em analogia com um "pipeline" de instruções).

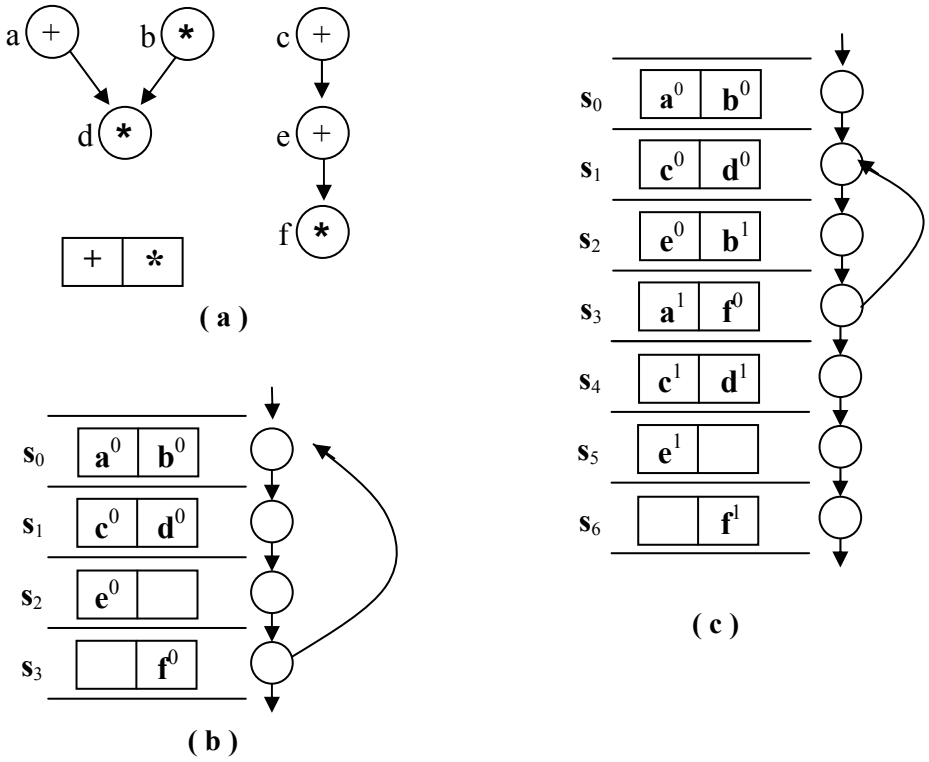


Figura 2.1: Exemplo ilustrativo de um escalonamento.

O corpo do laço paralelizado é denominado de *kernel* (corresponde aos estados s_1, s_2 e s_3). Por um lado, é preciso que algumas operações sejam previamente executadas para que produzam os valores que serão consumidos pelas instâncias de operações no "kernel". Tais instâncias constituem o chamado *pre-amble* (corresponde ao estado s_0). Terminadas todas as iterações do "kernel", restam algumas instâncias de operações que não foram ainda executadas. Estas instâncias que precisam ser executadas para completar iterações não concluídas no "kernel" constituem o chamado *post-amble* (corresponde aos estados s_4, s_5 e s_6).

A partir da existência do "kernel", surgem as noções de intervalo de introdução de dados ou *dii* ("data-introduction interval") e a latência. O *intervalo de introdução de dados* é o número mínimo de ciclos necessários entre a execução de sucessivas iterações do laço [SANT98]. A *latência* é o intervalo entre o tempo de chegada dos dados de entrada e o tempo que os respectivos dados de saídas são gerados [SANT98]. Por exemplo, para o escalonamento da figura 2.1c, o valor de dii é igual a 3 ciclos e o valor da latência é igual a 5 ciclos. Já na figura 2.1b, tanto a latência quanto o dii tem valores iguais a 4 ciclos, já que o laço não está paralelizado.

As noções exemplificadas acima são em seguida formalizadas. As definições abaixo são utilizadas na modelagem do problema de escalonamento com paralelização de laços.

Definição 2.1 – Um *grafo polar de fluxo de dados* DFG(V, E) é um grafo orientado onde cada vértice $v_i \in V$ representa uma operação e onde cada aresta $(v_i, v_j) \in E$ representa uma dependência de dados entre as operações v_i e v_j. Os pólos são os vértices v_0 e v_n, denominados fonte e sumidouro.

Definição 2.2 - Um *grafo polar da máquina de estados* SMG(S, T) é um grafo orientado onde cada vértice $s_i \in S$ representa um estado e cada aresta $(t_i, t_j) \in T$ representa uma transição entre estados s_i e s_j. Os pólos são os vértices s_0 e s_n, denominados fonte e sumidouro.

Definição 2.3 - Um *grafo da unidade operativa* DPG(C, W) é um grafo não orientado onde cada vértice $c_i \in C$ representa um componente onde cada aresta $(w_i, w_j) \in W$ representa uma interconexão entre os componentes c_i e c_j.

Definição 2.4 - Um *vetor de restrições de recursos* **a** é um vetor onde cada componente a_k representa o número de unidades funcionais disponíveis de um determinado tipo $k \in \{1, 2, ... , R\}$.

Definição 2.5 – Uma função τ: $V \rightarrow \{1, 2, ... ,R\}$ é uma função que mapeia cada operação para um tipo de recurso $k \in \{1, 2, ... , R\}$ onde será executada.

Definição 2.6 – O *atraso de execução* d_i é o valor correspondente ao número de ciclos necessários para completar a execução de uma operação v_i em um recurso do tipo $\tau(v_i)$.

Definição 2.7 – Uma *instância* de uma operação v_i na iteração m do laço, denotada por v_i^m, é uma réplica daquela operação que consome os valores dos operandos disponíveis na m-ésima iteração no laço.

Definição 2.8 – Dado um caminho $p = \langle s_1, s_2, s_3, ... s_k \rangle$ no SMG, o *comprimento* do caminho p é o número k de estados nele incluídos.

Definição 2.9 – Seja N o número de iterações de um laço. Dado o conjunto de instâncias $I = \{ v_i^n \mid v_i \in V \text{ e } n \leq N$. Uma função denominada *escalonamento* $\varphi: I \rightarrow S$ é uma função que mapeia cada instância v_i^n de I para um estado $s_k = \varphi(v_i^n)$ do SMG, tal que:

- $\forall(v_i, v_j) \in E: (s_k = \varphi(v_i^n) \text{ e } s_r = \varphi(v_j^n)) \Rightarrow r \geq k + d_i$ (restrição de precedência) e
- $| \{v_i: \tau(v_i) = p \text{ e } k \leq m < k + d_i\} | \leq a_p$, para cada estado s_m, com m = 1, 2, ..., M e para cada tipo de recurso p = 1, 2, ..., R (restrição de recursos).

É preciso fazer uma ressalva em relação à definição de escalonamento de um laço. A Definição 2.9 assume como restrições de precedência somente as dependências entre operações pertencentes a uma *mesma* iteração do laço. Em outras palavras, a Definição 2.9 não inclui as dependências entre instâncias pertencentes a iterações distintas (*loop-carried dependences*). Esta exclusão é deliberada, uma vez que este trabalho não aborda tais dependências, devido à limitação de tempo. Em conseqüência, os "benchmarks" escolhidos para os experimentos não contém esse tipo de dependências. O suporte para a dependência é proposto como trabalho futuro, conforme será discutido no Capítulo 5.

Parte da modelagem utilizada neste trabalho baseia-se em noções decorrentes de caminhos em grafos, como formalizado a seguir.

É importante notar que os pólos fonte e sumidouro do DFG representam as entradas e saídas do sistema, respectivamente. Portanto, ao se escalonar um vértice fonte u do DFG em um estado φ(u) do SMG, este estado corresponde ao instante em que os valores nas entradas estão disponíveis (amostragem das entradas). Analogamente, ao se escalonar um vértice sumidouro v do DFG em um estado φ(v) do SMG, este estado corresponde ao instante em que os valores para as saídas já foram calculados (disponibilidade das saídas). Por isso, a caracterização do comportamento temporal de laços em termos de valores de entrada e saída será feita medindo-se a distância entre tais arestas do SMG.

Note também que, se um DFG representa o corpo de um laço que é executado N vezes, a instância do vértice fonte da iteração i (v^i), representa os valores de entrada consumidos na i-ésima iteração. Assim, u^i e u^j representam valores de entrada, consumidos nas iterações i e j do laço, respectivamente.

Definição 2.10 – Dado um grafo orientado G(V, E) e dois vértices arbitrários v_i e $v_j \in V$, o vértice v_i alcança v_j através de p, escrito $v_i \xrightarrow{p} v_j$, se há um caminho p de v_i até v_j. Muitas vezes o caminho não necessita ser nomeado, escrito $v_i \xrightarrow{*} v_j$. Note que este caminho pode ser trivial caso $v_i = v_j$.

Definição 2.11 – Dados um grafo orientado G(V, E) e dois vértices arbitrários v_i e $v_j \in V$ tais que $v_i \xrightarrow{p} v_j$, a *distância* entre os vértices v_i e v_j, denotada por $\delta(v_i, v_j)$, é igual ao número de arestas no caminho p.

Definição 2.12 – Sejam s_0, ..., s_i, ... , s_j, ... , s_n estados sucessivos do SMG. A transição (s_j, s_i), que define o corpo de um laço cujo primeiro estado é s_i e cujo último estado é s_j, é denominada *back edge*.

Um laço costuma ser caracterizado pelo tempo gasto em uma iteração e pelo intervalo entre o início de iterações sucessivas, conforme formalizado a seguir.

Definição 2.13 – Sejam u e v os pólos fonte e sumidouro, respectivamente, do DFG que representa o corpo de um laço. A *latência da n-ésima iteração* do laço, denotada por $\lambda(n)$, é igual a $\delta(\varphi(u^n), \varphi(v^n))$.

Definição 2.14 – A *latência de um laço* com N iterações, denotada por λ, é igual a $\max\{\lambda(n)$, com $n = 0, 1, ..., N-1\}$.

Definição 2.15 – Seja u o pólo fonte de um DFG que representa o corpo de um laço. O intervalo de introdução de dados entre as iterações n e n+1, denotado por dii (n, n+1) é igual a $\delta(\varphi (u^n), \varphi (u^{n+1}))$.

Definição 2.16 – O *intervalo de introdução de dados* de um laço com N iteraçcões, denotado por dii, é igual a max { dii (n, n+1), com n= 0, 1, ..., N-2 }.

Durante o escalonamento instâncias são associadas a estados. As instâncias passíveis de escalonamento em um dado estado são aquelas que ainda não foram escalonadas e cujos predecessores foram todos escalonados há tempo suficientemente longo para acomodar seus atrasos de execução, conforme formalizado a seguir.

Definição 2.17 – Dado um SMG e um estado arbitrário $s_k \in S$, o conjunto das *instâncias disponíveis* em s_k, denotado por A_k, é o conjunto de todas as instâncias v_j^x, tais que:

- A instância v_j^x não foi escalonada em s_k ou em algum estado s_m tal que $s_m \xrightarrow{*} s_k$.
- Para toda instância v_i^x escalonada em um estado arbitrário s_p, tal que v_i é predecessor imediato de v_j, vale a desigualdade $\delta(s_p, s_k) \geq d_j$.

Nem todas as instâncias podem ser executadas em um único ciclo. Por isso, se uma instância v_j^x, com $d_j > 1$, for executada em um estado s_m ela não estará finalizada no estado s_k, caso a distância entre s_m e s_k for menor do que d_j. Esta noção é formalizada a seguir:

Definição 2.18 – Dado um SMG e um estado arbitrário $s_k \in S$, o conjunto das *instâncias pendentes* em s_k, denotado por U_k, é o conjunto de todas as instâncias v_j^x, tais que:

- A instância v_j^x já foi escalonada em um estado arbitrário s_m tal que $s_m \xrightarrow{*} s_k$.
- $\delta(s_m, s_k) < d_j$.

Conforme será mostrado no Capítulo 4, o algoritmo utilizado neste trabalho para realizar a paralelização de laços, baseia-se na noção de estados equivalentes, que é formalizada a seguir.

Definição 2.19 – Dado um SMG e dois estados arbitrários s_i e $s_j \in S$, s_i é equivalente a s_j, escrito $s_i \equiv s_j$, se e somente $A_i = A_j$ e $U_i = U_j$.

Definição 2.20 – Uma *janela* de tamanho W começando em uma dada iteração i é o conjunto de iterações compreendidas entre a iteração i a iteração i + W - 1.

2.2 Escalonamento com Restrição de Recursos

Informalmente, o problema consiste em utilizar as técnicas de paralelização de laços (por exemplo, "loop pipelining" e "loop unrolling") no escalonamento das operações de acordo com a quantidade de recursos disponíveis (restrição de recursos), observando restrições de precedência (produção e consumo de dados), de forma a minimizar o intervalo de iniciação do laço ou sua latência.

Antes de formalizar o problema de escalonamento com restrição de recursos, serão mostrados exemplos de como as técnicas de paralelização se interam ao escalonamento.

2.2.1 Exemplo de "loop pipelining"

A Figura 2.2 mostra um trecho de descrição comportamental com 2 multiplicadores e 3 operações lógico-aritméticas, caracterizando assim os dois tipos de recursos: multiplicador (MULT) e unidade lógica e aritmética (ULA). Alguns valores produzidos

são consumidos por outras operações, fazendo com que exista uma *restrição de precedência*.

Assuma que o laço execute um número infinito de iterações. Por isso, não se representa o teste para controlar a saída do laço. Assuma também que a execução de uma operação executada no recurso ALU leva um único ciclo e a execução de uma operação executada em um MULT leva dois ciclos.

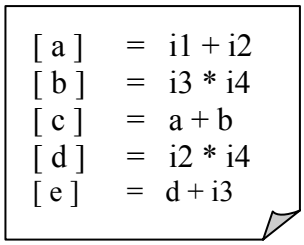
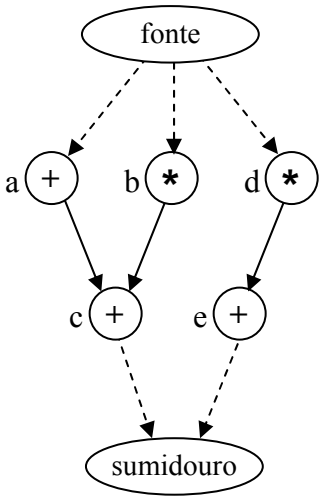


Figura 2.2: **Trecho de descrição comportamental.**

A Figura 2.3a mostra o DFG obtido a partir desta descrição comportamental. Os pólos do DFG, denominados de fonte (“source”) e sumidouro (“sink”), servem apenas para a identificação das entradas e saídas primárias e têm atraso de execução igual a zero. As arestas pontilhadas com origem no vértice fonte representam os valores que estão disponíveis inicialmente (entradas primárias) para cada iteração do laço. Similarmente, as arestas pontilhadas incidentes no vértice sumidouro representam os valores que estão disponíveis ao final da execução de uma iteração do laço.

No exemplo da Figura 2.3, assume-se uma regra de prioridade para selecionar instâncias que competem pelo mesmo recurso, a qual é codificada na forma de uma permutação de instâncias dentro de uma dada janela de iterações. A Figura 2.3b contém uma codificação de prioridades, que é uma permutação $\prod$ de instâncias das operações com tamanho de janela igual a dois (W = 2). Na permutação $\prod$, i corresponde à menor iteração atualmente dentro da janela. A Figura 2.3c indica que as operações “a”, “c” e “e” serão mapeadas para o somador, enquanto que as operações “b” e “d” serão mapeadas para o multiplicador. Note que há apenas dois recursos: um multiplicador

(MULT) e uma unidade lógico-aritmética (ULA), o que representa uma restrição de recursos.



(a)

$\Pi = \{ b^i, d^i, b^{i+1}, d^{i+1}, a^i, e^i, e^{i+1}, c^i, c^{i+1}, a^{i+1} \}$

(b)

W = 2

$\tau(b) = \tau(d) = MULT$
$\tau(a) = \tau(c) = \tau(e) = ULA$
n (MULT) = 1
n (ULA) = 1

(c)

Figura 2.3: **DFG extraído da descrição comportamental da Figura 2.2.**

A seguir, será descrito o processo de construção do SMG, estado por estado, para o exemplo da Figura 2.3. O processo é ilustrado nas Figuras 2.4 e 2.5. Os conjuntos de instâncias disponíveis e instâncias pendentes para cada estado são mostrados na Tabela 2.1. Em particular, dado o estado s_0, o conjunto A_0 contém instâncias de operações que dependem somente de valores de entrada, enquanto que, o conjunto U_0 está vazio, pois nenhuma instância foi escalonada antes do estado atual (s_0). Esses conjuntos possuem instâncias ordenadas de acordo com a codificação de prioridades Π.

Estado atual: s_0 (Figura 2.4a)

Neste primeiro estado estão disponíveis operações da iteração 0 e 1 do laço (W = 2 e menor iteração = 0) e nenhuma instância está pendente. De acordo com A_0 (conjunto de instâncias disponíveis), as operações escalonadas no estado s_0 são a^0, mapeada para o ALU, e b^0, mapeada para a MULT. Como a instância b^0 é um multiplicador, ela devera ser escalonada no próximo estado, pois leva dois ciclos para executar. Assim, um novo estado s_1 é criado para acomodar a execução das próximas instâncias e a instância b^0 é inserida em U_1 (instância pendente).

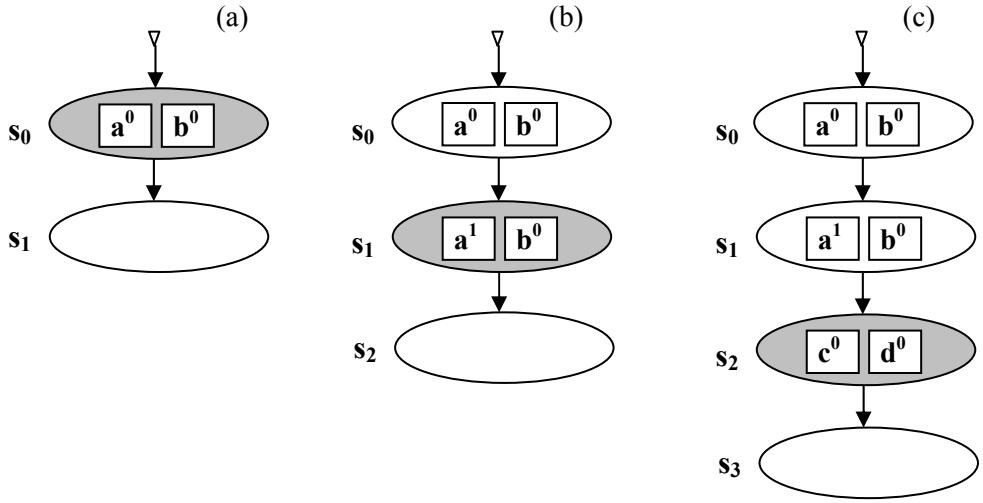


Figura 2.4: **Escalonando o laço da Figura 2.2 ("loop pipelining" - Parte I).**

Estado atual: s_1 (Figura 2.4b)

Como a instância b^0 começou a executar no estado anterior e está pendente, ela deve ser escalonada neste estado. As instâncias escalonadas são b^0 (pendente) e a^1 (disponível) mapeadas para MULT e ULA, respectivamente. Com o fim do escalonamento da instância b^0, a instância c^0 fica disponível para o estado s_2 e, por isso, passa a ser um elemento do conjunto A_2.

Estado atual: s_2 (Figura 2.4c)

Embora as instâncias b^1 e d^1 estejam disponíveis e competem pelo multiplicador, a instância d^0 tem maior prioridade, por isso ela é escalonada neste estado e fica pendente para continuar sua execução no próximo estado. A instância c^0 também é escalonada.

Estado atual: s_3 (Figura 2.5a)

Aqui é escalonada somente a instâncias d^0 para executar no recurso do tipo MULT (segundo ciclo da instância). Com o escalonamento da instância d^0 neste estado, a instância e^0 fica disponível para o próximo estado. Apesar de existir um recurso do tipo ALU livre, não existe nenhuma instância disponível que possa ser escalonada nesse estado.

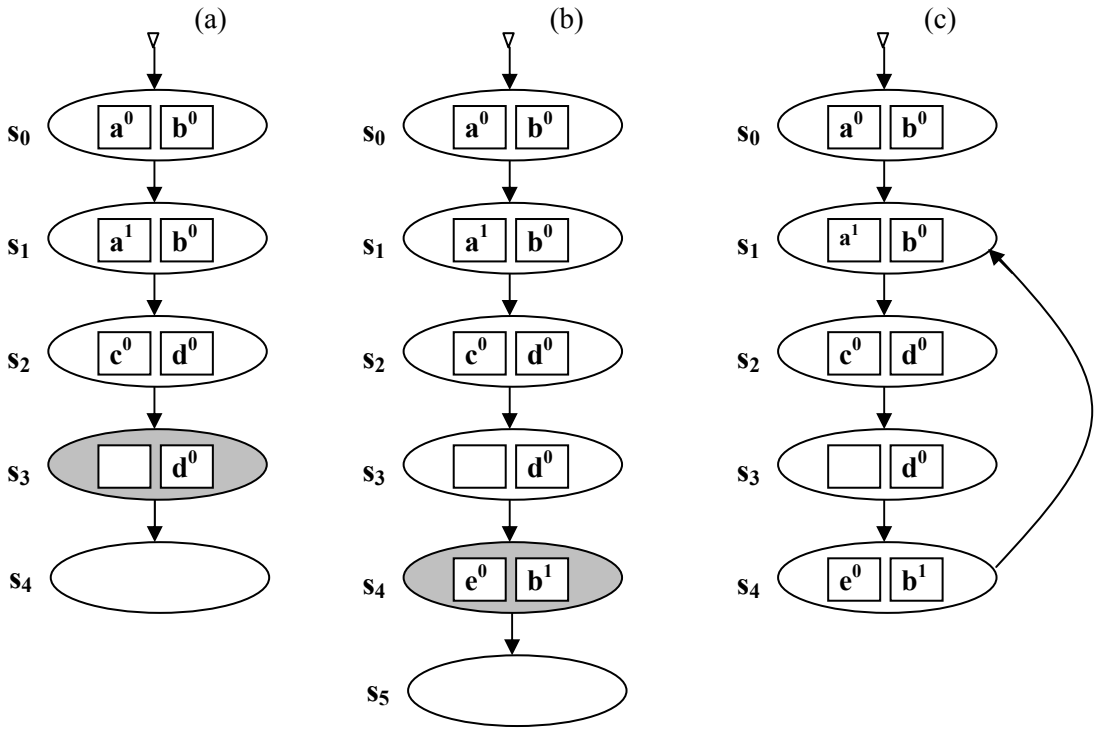


Figura 2.5: **Escalonando o laço da Figura 2.2 ("loop pipelining" - Parte II).**

Estado atual: s_4 (Figura 2.5b)

Neste estado, e^0 e b^1 são as instâncias selecionadas. Como b^1 leva dois ciclos para terminar de executar, ela deverá ser escalonada no próximo estado e por isso é inserida em U_5. Com o escalonamento da instância e^0 neste estado, todas as operações da iteração 0 foram executadas. Como a menor iteração é a 1, a iteração 2 do laço deve começar a executar para manter o tamanho da janela em 2 iterações, ou seja, a janela avança de uma iteração. Assim, as instâncias a^2 e b^2 tornam-se elementos de A_5.

Tabela 2.1: **Os conjuntos A_k para os exemplos nas Figuras 2.4 e 2.5.**

Estado	Instâncias disponíveis	Instâncias pendentes
s_0	$A_0 = \{ b^0, d^0, b^1, d^1, a^0, a^1 \}$	$U_0 = \emptyset$
s_1	$A_1 = \{ d^0, b^1, d^1, a^1 \}$	$U_1 = \{ b^0 \}$
s_2	$A_2 = \{ d^0, b^1, d^1, c^0 \}$	$U_2 = \emptyset$
s_3	$A_3 = \{ b^1, d^1 \}$	$U_3 = \{ d^0 \}$
s_4	$A_4 = \{ b^1, d^1, e^0 \}$	$U_4 = \emptyset$
s_5	$A_5 = \{ d^1, b^2, d^2, a^2 \}$	$U_5 = \{ b^1 \}$

Estado atual: s_5 (Figura 2.5c)

Note que A_5 coincide com A_1 exceto pelos números das iterações das instâncias, ou seja, as mesmas operações disponíveis em A_5 estão disponíveis em A_1 com uma iteração de atraso. Note também que U_1 e U_5 contém operações idênticas, exceto pelos números das instâncias. Quando isso acontece, diz-se que um *estado equivalente* foi detectado (veja Definição 2.19). Tal equivalência significa que o escalonamento dos estados s_5 e s_1 e de seus respectivos sucessores resultaria em idênticas seqüências periódicas de operações, que só se distinguem por pertencerem a diferentes iterações. Assim, ao invés de escalonar o estado s_5, insere-se uma transição do estado s_4 diretamente para o estado s_1, sendo o estado s_5 removido. Note que a transição (s_4, s_1) é uma aresta do tipo "back edge" (veja Definição 2.12), o que caracteriza o fechamento do laço paralelizado.

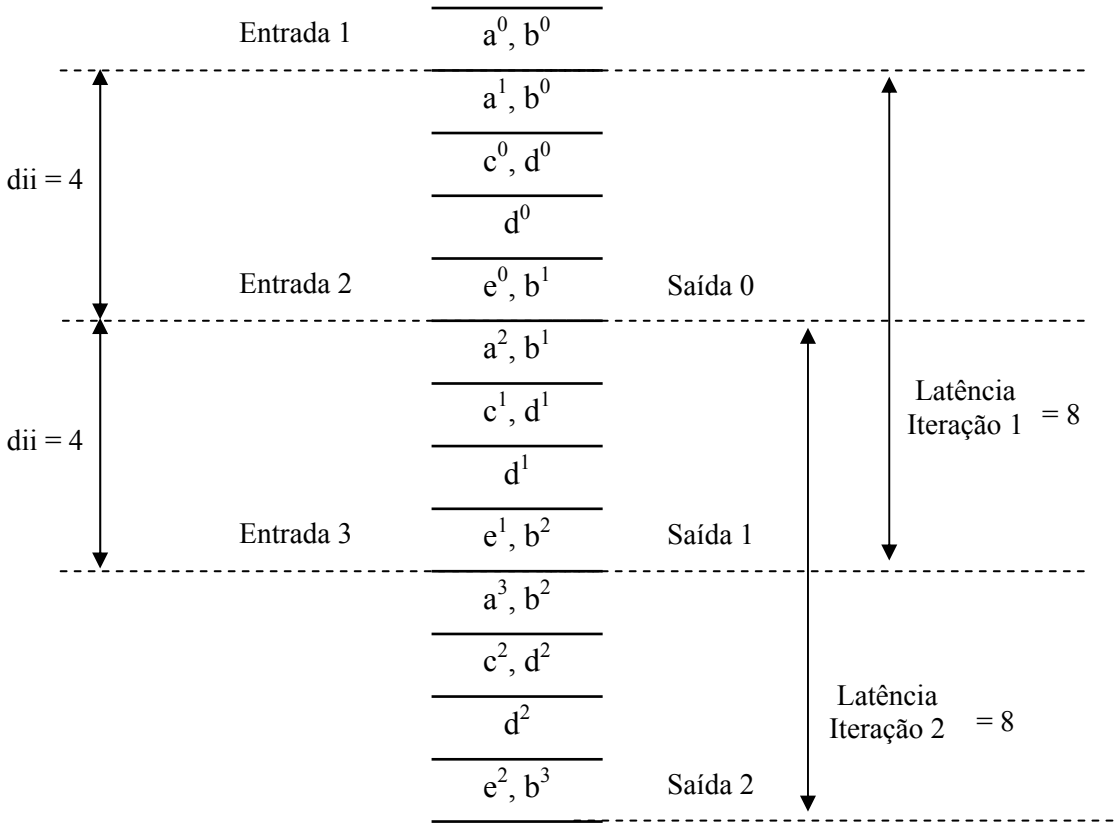


Figura 2.6: Seqüência de dados de entrada e de saída para o SMG da Figura 2.5c.

Note que o SMG final na Figura 2.5c contém um ciclo. Este ciclo possui instâncias de operações de diferentes iterações do laço, que está conseqüentemente paralelizado. Na figura, o "pre-amble" é automaticamente separado do "kernel" como resultado da

inserção da aresta do tipo “back-edge” (o mesmo aconteceria com o “post-amble” caso não fosse assumido um número infinito de iterações).

A Figura 2.6 mostra algumas das seqüências de dados que envolvem a execução do algoritmo. As setas indicam o tempo de chegada de dados de entradas para cada nova iteração do corpo do laço original (a partir da iteração 1) e indicam também o tempo onde os dados das saídas estão disponíveis. Note que, embora as operações do laço sejam exatamente as mesmas a cada iteração, os valores por elas consumidos são distintos, pois são função das novas entradas amostradas a cada novo intervalo de iniciação.

2.2.2 Exemplo de “loop unrolling”

Esta seção apresenta o escalonamento gerado via “loop unrolling” do DFG mostrado na Figura 2.3a. Para este exemplo, assume-se um tamanho de janela igual a dois e uma nova codificação de prioridade Π mostrada na Figura 2.7a. A Figura 2.7b indica que as operações “b” e “d” serão mapeadas para o multiplicador enquanto que as operações “a”, “c” e “e” serão mapeadas para a unidade lógico-aritmética. Note que há 4 recursos: dois multiplicadores (MULT) e duas unidades lógico-aritméticas (ULA), o que representa a restrição de recursos para esse exemplo. A Figura 2.7c mostra o “kernel” do escalonamento enquanto que os conjunto de instâncias disponíveis e instâncias pendentes para cada estado são mostrados na Tabela 2.2. É importante frisar que esses conjuntos possuem instâncias ordenadas de acordo com a codificação de prioridades Π.

Tabela 2.2: Os conjuntos A_k para o exemplo na Figura 2.7.

Estado	Instâncias disponíveis	Instâncias pendentes
s_0	$A_0 = \{ a^0, b^0, b^1, a^1, d^0, d^1 \}$	$U_0 = \emptyset$
s_1	$A_1 = \{ d^0, d^1 \}$	$U_1 = \{ b^0, b^1 \}$
s_2	$A_2 = \{ c^1, c^0, d^0, d^1 \}$	$U_2 = \emptyset$
s_3	$A_3 = \{ d^0, d^1 \}$	$U_3 = \emptyset$
s_4	$A_4 = \{ e^0, e^1 \}$	$U_4 = \emptyset$

Observe que o corpo do laço escalonado contém mais de uma instância da mesma operação, como por exemplo: as instâncias a^0 e a^1 da operação “a”, que foram

escalonadas no estado s_0. Quando isso ocorre diz-se que o laço foi gerado via *loop unrolling*. Neste caso, ao invés de um "kernel pipelinizado", obtém-se um laço cujo corpo corresponde ao corpo do laço original repetido várias vezes.

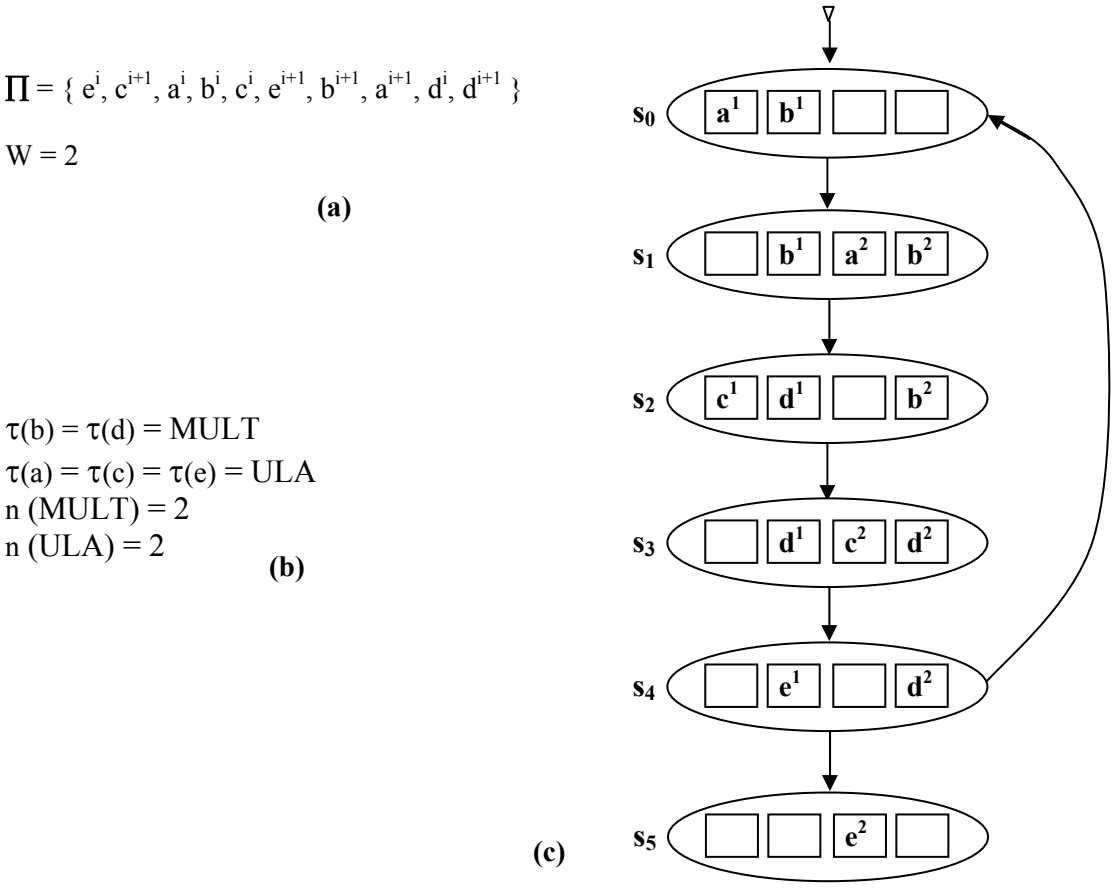


Figura 2.7: Escalonamento do laço da Figura 2.2 ("loop unrolling").

2.2.3 Formulação do problema

Formalmente, o escalonamento sob restrição de recursos pode ser formulado como um problema de otimização combinatória. Esta dissertação trata dos problemas formalizados a seguir:

Problema 2.1: Dado um grafo de fluxo de dados DFG(V, E) e um vetor de restrição de recursos **a**, encontre um escalonamento φ que minimize o intervalo de introdução de dados dii.

Problema 2.2: Dado um grafo de fluxo de dados DFG(V, E) e um vetor de restrição de recursos **a**, encontre um escalonamento φ que minimize a latência λ.

Para abordar os problemas acima serão utilizadas as técnicas de "loop pipelining" e "loop unrolling", ilustrados na seção anterior. Tais técnica, por aumentarem a exposição de paralelismo, permitem uma melhor utilização dos recursos. Como conseqüência, um menor dii e uma menor latência podem ser obtidos.

Essa abordagem será mostrada no próximo capítulo. Antes, porém, será apresentada a seguir uma breve revisão da literatura correlata.

2.3 Breve Revisão Bibliográfica

Paralelização de laços é uma idéia relativamente antiga. Atualmente, há vários algoritmos e modelos para paralelização. Dentre outros, podemos citar *Modulo Scheduling* [MLAN88] e *Enhanced Pipeline Scheduling* [EBCI87].

Modulo Scheduling ("MS") é uma das mais populares técnicas de paralelização de laços e, conseqüentemente, é utilizada como base para outros algoritmos. MS assume um dii fixo para o corpo do laço, que é calculado antecipadamente.

Enhanced Pipeline Scheduling ("EPS") modela o laço como uma seqüência cíclica de operações. O laço é paralelizado movendo operações no sentido contrário ao do ciclo, o que equivale a obter instâncias das operações movidas em iterações anteriores.

Outra alternativa é utilizar loop unrolling [HENN96] como mecanismo básico. *Perfect Pipelining* ("PP") [AIKE88] trabalha da seguinte maneira: o laço é desenrolado um certo número de vezes e a partir daí, as operações são escalonadas até que um padrão seja detectado para o laço (este é justamente o mecanismo que foi utilizado no exemplo da Seção 2.2.1). Algumas regras são necessárias para garantir que se alcance um padrão. Uma desvantagem do PP é que em alguns casos, a convergência para um padrão pode

ser muito demorada. Uma vantagem do PP é que a estrutura do algoritmo é tal que o procedimento de paralelização é separado do procedimento escalonador, o que evita que as heurísticas de escalonamento limitem o processo de paralelização.

Uma revisão mais detalhada dessas técnicas pode ser encontrada em [SANT00].

Com a especificação de sistemas embutidos com restrições de tempo e de recursos cada vez mais severas, torna-se importante escolher um algoritmo de paralelização de laços adequado à exploração do espaço de projeto (“design space exploration”). Isto motiva uma abordagem orientada à exploração de soluções alternativas. Tal abordagem é descrita no próximo capítulo.

Capítulo

3 Uma Abordagem Orientada à Exploração Automática

Este capítulo descreve uma abordagem para a resolução do problema definido na Seção 2.2.3, através da exploração automática de soluções alternativas, conforme proposto em [SANT98].

A Seção 2.3 descreve alguns algoritmos, dentre os vários existentes na literatura, utilizados para resolver o problema de paralelização de laços. Em sua maioria, esses algoritmos possuem baixa complexidade, mas por utilizarem diferentes heurísticas, geralmente conflitantes, e gerarem uma única solução, não conseguem gerar soluções de boa qualidade sob restrições muito severas. Por outro lado, existem algoritmos que geram ótimas soluções, mas tornam-se inviáveis por apresentarem complexidade exponencial.

Para aumentar as chances de se encontrar uma boa solução sem utilizar algoritmos muito complexos, torna-se necessário não restringir o espaço de busca a uma única solução, ou seja, explorar diversas soluções na tentativa de otimizar os resultados, sem cair em uma busca exaustiva. Neste trabalho, busca-se um compromisso entre o tempo de busca e a qualidade da solução obtida.

Torna-se necessário também identificar um método de escalonamento que possa ser facilmente generalizado, agregando um tratamento eficiente de execuções condicionais.

É possível explorar soluções alternativas com a definição de uma codificação de prioridades para determinar em que ordem as operações disponíveis serão escalonadas [SANT98]. A idéia-chave da abordagem de exploração de soluções é a seguinte: diferentes codificações de prioridades resultam em diferentes soluções com custos possivelmente distintos. Dessa forma, a otimização do custo é suportada pela monitoração dos custos de diferentes soluções exploradas (diferentes codificações) e a

escolha da solução de menor custo. Podem ser usados vários métodos para gerar codificações de prioridade de forma a conduzir a busca para soluções cada vez melhores, mas isto é objeto de um outro trabalho de pesquisa no âmbito do projeto OASIS. A codificação de prioridade usada neste trabalho baseia-se em permutações de instâncias das operações do DFG, conforme será descrito na Seção 3.2.

Nesse contexto, o algoritmo proposto por Aiken, Nicolau e Novack [AIKE95] parece suficientemente geral, garantindo a flexibilidade necessária para o tratamento de condicionais e laços, podendo ser facilmente adaptado para explorar soluções alternativas, conforme mostrado em [SANT98]. Por estas razões, tal algoritmo será adotado neste trabalho, conforme descrito no Capítulo 4.

3.1 A Decomposição da Abordagem

A abordagem utilizada para resolver o problema de paralelização de laços definido na Seção 2.2 pode ser resumida como a interação entre dois blocos principais: o *explorador* e o *construtor*.

Uma visão geral dessa abordagem é mostrada na Figura 3.1. Pode-se observar nesta figura a independência entre os blocos e a comunicação entre os mesmos através dos parâmetros $\prod$ e custo.

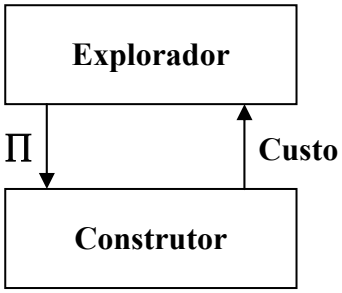


Figura 3.1: **Visão geral da abordagem.**

O explorador é o bloco encarregado da criação da codificação de prioridade das instâncias de operações a serem escalonadas. Essa codificação é definida como uma

permutação ∏ de instâncias das operações do DFG. A ordem das instâncias em ∏ define a prioridade entre as mesmas. Assim, no caso de mais de uma instância estar disponível para escalonamento, a prioridade em ∏ determina qual das instâncias é selecionada.

O construtor tem a função de gerar uma solução a partir de uma permutação ∏ obtida do explorador, através do escalonamento das instâncias de operações do DFG e pelo cálculo do custo da solução. Neste trabalho, o custo da solução é o dii (intervalo de introdução de dados) ou a letência. Note que o construtor limita-se a seguir as prioridades codificadas em ∏; ele não altera essa permutação, pois somente o explorador pode fazê-lo.

Uma grande vantagem dessa abordagem é a separação das funcionalidades em blocos diferentes. Isso torna a abordagem mais flexível, permitindo a adaptação do algoritmo em um dos blocos sem interferir na funcionalidade do outro. Por exemplo, pode ser utilizado um construtor que execute um escalonamento simples ou um construtor que utilize técnicas de paralelização de laços para escalonar suas operações. Para isso, é necessário apenas manter a interface entre os blocos e trocar-se o construtor.

Outra vantagem dessa abordagem é a facilidade de controlar o número de soluções que se deseja explorar, através de um parâmetro definido pelo usuário que limite o número de codificações de prioridade geradas. Dessa forma, é provável alcançar-se um compromisso entre a qualidade de uma solução e o esforço computacional para obtê-la.

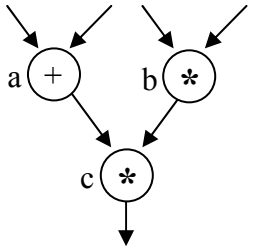
A seguir, a abordagem é descrita mais detalhadamente. A Seção 3.2 explica a codificação de prioridades e a Seção 3.3 mostra os sub-blocos que fazem parte do construtor.

3.2 A Codificação de Prioridade

Como já mencionado anteriormente, a codificação de prioridade é definida como uma permutação ∏ de instâncias das operações do DFG e é tarefa exclusiva do explorador.

Ao construir a permutação ∏, através de um critério de busca local, o explorador não considera as restrições de precedência definidas no DFG, pois isto é tarefa do construtor. O explorador limita-se a obter uma ordenação para todas as instâncias de operações do DFG dentro de uma janela determinada.

A Figura 3.2 mostra o exemplo de uma permutação ∏. Dado o DFG e o tamanho da janela (W=2), a permutação ∏ é composta por instâncias das operações do DFG da iteração i e da iteração i+1.



W = 2

$\prod = \{ a^i, b^i, c^i, a^{i+1}, b^{i+1}, c^{i+1} \}$

Figura 3.2: **Exemplo de codificação de prioridade.**

A codificação de prioridade é utilizada pelo construtor somente como critério de desempate na seleção de instâncias. Um critério de desempate é necessário quando o conjunto de instâncias disponíveis para o escalonamento em um dado estado for maior que o número de módulos disponíveis naquele estado. Assim, a permutação ∏ determina qual ou quais das instâncias disponíveis serão escolhidas para serem escalonadas nesse estado. A instância ocupando a posição de mais alta prioridade em ∏ é a selecionada. Uma instância é considerada disponível quando todas as operações predecessoras na mesma iteração já tenham sido escalonadas em estados anteriores e há tempo suficientemente longo para acomodar seus atrasos de execução (vide Definição 2.17).

Dado um DFG = (V, E), a codificação de prioridade determinada por essa abordagem é a permutação ∏ das instâncias das operações V. A noção de prioridade é associada com a respectiva posição na permutação. Suponha que v_i^m seja a instância da operação v_i na m-ésima iteração, $\prod(v_i^m)$ denota a posição de uma instância v_i^m na permutação ∏. Diz-

se que v_i^m precede v_j^m na permutação Π, matematicamente escrito como $v_i^m \prec_\Pi v_j^m$, se $\Pi(v_i^m) < \Pi(v_j^m)$.

3.3 O Construtor de Soluções

O construtor consiste em dois blocos principais: um *escalonador* e um *paralelizador*, como mostra a Figura 3.3. Pode-se observar na figura a independência entre esses dois blocos e os parâmetros de comunicação entre eles.

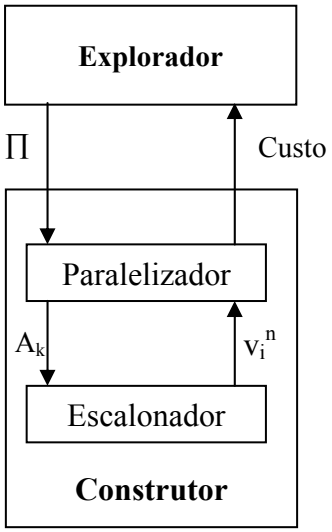


Figura 3.3: **Detalhamento do Construtor.**

O paralelizador tem a função de capturar o paralelismo entre as operações da descrição comportamental e utilizá-lo na paralelização de laços. Inicialmente, o paralelizador cria um estado atual (s_k) onde as instâncias serão escalonadas. O conjunto de instâncias disponíveis para serem escalonadas em um estado s_k é denotado por A_k.

Existindo instâncias disponíveis em A_k e módulos para o escalonamento dessas instâncias, o paralelizador envia o conjunto A_k para o escalonador. Este, baseado na codificação de prioridade, determina qual a instância será escalonada e retorna esta instância para o paralelizador. Sabendo qual a instância dentre as disponíveis deverá ser escalonada, o paralelizador marca esta instância como escalonada e atualiza o conjunto A_k. Isso se repete até não existirem mais instâncias passíveis de escalonamento. Então, o

paralelizador cria um novo estado s_{k+1}, chamado de próximo estado e atualiza o seu respectivo conjunto A_{k+1}. Dessa forma, o SMG vai sendo construído pelo paralelizador passo a passo.

O escalonamento chega ao fim quando o paralelizador detectar a existência de um conjunto A_p equivalente à A_k (atual) e um conjunto U_p equivalente ao U_k. Isto significa que existe um estado s_p equivalente ao atual (veja Definição 2.19). Como o escalonamento a partir de s_k seria equivalente ao escalonamento a partir de s_p, ao invés de escalonar s_k, o paralelizador insere uma aresta do tipo "back edge" de s_{k-1} a s_p. Dessa forma, o paralelizador torna cíclico o SMG, o que representa a repetição de uma seqüência idêntica de instâncias de operações pertencentes a iterações distintas, ou seja, representa um laço paralelizado.

Capítulo

4 Implementações e Experimentos

Este capítulo descreve a parte prática deste trabalho, mostrando a implementação da abordagem introduzida e definida nos Capítulos 2 e 3, bem como os experimentos realizados e os resultados obtidos da abordagem.

4.1 Plataforma de Trabalho

O protótipo que captura a abordagem mostrada no Capítulo 3 foi desenvolvido no âmbito do assim-chamado projeto OASIS: Modelagem, Síntese e **O**timização de **A**rquiteturas para **SIS**temas Digitais. Este projeto tem como plataforma-alvo de trabalho um microcomputador PC, sob o sistema operacional Linux e usando o ambiente kde [KDE01]. A linguagem adotada para todas as ferramentas desenvolvidas no âmbito do projeto é C++. Algumas classes básicas como Heap, List, etc. são retiradas de [WEIS96], com algumas adaptações. Outras classes utilizadas são de uma Interface Orientada a Objetos para Síntese de Alto Nível, atualmente sendo desenvolvida pelo grupo do projeto OASIS. Outras classes foram criadas especificamente para este trabalho. Utiliza-se o pacote daVinci [FROL96] para a visualização dos grafos que modelam o problema e as soluções.

4.2 O Algoritmo de Paralelização

Para resolver o problema formulado na Seção 2.2.2, adotou-se o algoritmo proposto por Aiken, Nicolau e Novack [AIKE95] adaptado em [SANT98] para a abordagem de exploração de soluções alternativas. Nossa implementação de tal algoritmo é descrita a seguir.

Conforme mencionado na Seção 2.2, o algoritmo de paralelização busca a minimização do dii (intervalo de introdução de dados) ou da latência, respeitando as restrições de

recursos (vetor **a**), de precedência (arestas do DFG) e obedecendo a uma dada codificação de prioridade (permutação $\prod$).

A estrutura principal do Algoritmo de Paralelização implementado é mostrada no Algoritmo 4.1, enquanto que os demais métodos utilizados por ele estão mostrados nos Algoritmos 4.2, 4.3, 4.4, 4.5, 4.6 e 4.7. Como tais métodos constroem o grafo SMG(S, T) a partir do grafo DFG(V, E), assume-se que S e T são variáveis visíveis a todos os métodos.

```
Escalonador (a, ∏, W, DFG(V, E))
    menor_it = 0;
    I = { v_i^m | v_i ∈ V ∧ 0 ≤ m < W};
    crie estado inicial s_0 no SMG;
    A_0 = { v_i^m ∈ I | v_o é o único predecessor de v_i no DFG};
    U_0 = ∅;
    próximo = s_0;
    enquanto (próximo ≠ nenhum) faça
        s_k = próximo;
        enquanto (∃ v_i^m ∈ I | m = menor_it) faça
            se (∃ s_j ∈ S | s_j ≡ s_k ) // Vide Definição 2.19
                RemovaEstado(s_k);
                InsiraBackEdge(s_k-1, s_j);
                dii = MaiorDii ( ); // Vide Definição 2.16
                próximo = nenhum;
            senão
                INST_k = EscaloneEstado( s_k, A_k, U_k, a, ∏, I );
                D = EncontreInstânciasDisponíveis(INST_k);
                P = EncontreInstânciasPendentes(INST_k);
                D = D ∪ A_k;
                s_k+1 = NovoEstado();
                A_k+1 = D;
                U_k+1 = P;
                próximo = s_k+1;
        se (próximo ≠ nenhum)
            menor_it = menor_it + 1;
            I = I ∪ { v_i^m | (v_i ∈ V) ∧ (m = menor_it + W - 1) };
    retorne (dii);
```

Algoritmo 4.1: Algoritmo de Paralelização.

Dada uma codificação de prioridade $\prod$ e um tamanho de janela (W), o Algoritmo 4.1 cria um SMG a partir do DFG, sob a restrição de recursos **a**, invocando o procedimento Escalonador (a, $\prod$, W, DFG(V, E)).

É o procedimento Escalonador (a, $\prod$, W, DFG(V, E)) que faz o controle do processo de escalonamento. Ele é responsável pela inicialização, pelo controle da janela (W) e pelo controle da detecção de estados equivalentes.

Após inicializar o conjunto I e a variável menor_it, o procedimento Escalonador (a, $\prod$, W, DFG(V, E)) cria um estado inicial e calcula as instâncias disponíveis para serem escalonadas dentro da janela inicial (W), ou seja, aquelas que têm como único predecessor o pólo fonte v_0.

O método Escalonador (a, $\prod$, W, DFG(V, E)) contém um laço principal que executa até que não existam mais estados para serem escalonados. Internamente a este, existe um outro laço que faz o controle da janela de iterações, ou seja, executa até que todas as operações da menor iteração tenham sido escalonadas. Quando isso ocorre, caso ainda exista algum estado para ser escalonado, instâncias de uma nova iteração são adicionadas ao conjunto I para conservar o tamanho da janela.

RemovaEstado(s_k)
    $S = S - \{ s_k \}$;
    $T = T - \{ (s_{k-1}, s_k) \}$;

InsiraBackEdge(s_{k-1}, s_j)
    $T = T \cup \{ (s_{k-1}, s_j) \}$;

Algoritmo 4.2: Métodos para remoção de estados e inserção de arestas.

Dentro do laço principal é feita a verificação de estados equivalentes. Se existir algum estado anteriormente escalonado equivalente ao estado atual, remove-se o estado atual (s_k) e insere-se uma aresta do tipo “back edge” de seu predecessor (s_{k-1}) ao estado a ele equivalente (s_j), e o dii é calculado através do Algoritmo 4.3. A remoção do estado atual é feita pelo método RemovaEstado(s_k), enquanto que, a inserção da aresta é feita pelo

método InsiraBackEdge(s_{k-1}, s_j) conforme descritos no Algoritmo 4.2. Caso não exista nenhum estado equivalente, um novo estado deve ser escalonado.

```
MaiorDII( )
    max_dii = ∞;
    para (n=0; n ≤ N-1; n = n+1)
        se dii (n, n+1) > max_dii
            então max_dii = dii (n, n_+1)
    retorne (max_dii);
```

Algoritmo 4.3: Método utilizado para encontrar o menor dii nas iterações.

O escalonamento das instâncias é feito pelo método EscaloneEstado(s_k, A_k, U_k, a, $\prod$, I), cujo comportamento é descrito no Algoritmo 4.6, retornando as instâncias escalonadas em $INST_k$.

O próximo passo é atualizar as instâncias disponíveis e encontrar as instâncias pendentes.

```
EncontreInstânciasDisponíveis (INST_k)
    para cada v_i^m ∈ INST_k
        para cada sucessor v_j^m de u^m no DFG
            se (todos os predecessores de v satisfazem Definição 2.16)
                D = D ∪ {v_j^m};
    retorne (D);

EncontreInstânciasPendentes (INST_k)
    P = ∅;
    para cada instância v_i^m ∈ INST_k
        se (v_i^m satisfaz Definição 2.17)
            P = P ∪ {v_i^m};
    retorne (P);
```

Algoritmo 4.4: Métodos para encontrar instâncias disponíveis e pendentes.

O Algoritmo 4.4 descreve os métodos EncontreInstânciasDisponiveis($INST_k$) e EncontreInstânciasPendentes($INST_k$). A partir das instâncias escalonadas ($INST_k$), o primeiro método verifica quais instâncias ficam disponíveis para o próximo estado, enquanto que o segundo seleciona as instâncias que foram escalonadas no estado atual e

ainda não terminaram de executar (necessitam de mais ciclos). Para que o escalonamento continue, um novo estado é criado através do método NovoEstado() conforme descrito no Algoritmo 4.5 e as instâncias disponíveis e pendentes lhe são atribuídas.

NovoEstado()
    Crie um estado s_{k+1};
    $S = S \cup \{ s_{k+1} \}$;
    $T = T \cup \{ (s_k, s_{k+1}) \}$;

Algoritmo 4.5: Método para criar um novo estado no SMG.

O Algoritmo 4.6 descreve o método EscaloneEstado(s_k, A_k, U_k, a, $\prod$, I) que, primeiramente escalona todas as instâncias pendentes (U_k) e depois chama o método SelecioneInstância(A_k, a, $\prod$), descrito no Algoritmo 4.7, que retorna a instância disponível de maior prioridade satisfazendo a restrição de recurso **a**. Enquanto existir uma instância para ser escalonada e recurso disponível para acomoda-la, o método EscaloneEstado(s_k, A_k, U_k, a, $\prod$, I) insere essa instância em $INST_k$ e a remove de A_k e de I. Quando todas as instâncias possíveis foram escalonadas em s_k, o métodos as retorna ($INST_k$).

EscaloneEstado (s_k, A_k, U_k, a, $\prod$, I)
    $INST_k = \varnothing$;
    para todo $v_i^m \in U_k$
        escalone v_i^m em s_k;
        $INST_k = INST_k \cup \{ v_i^m \}$;
    v_i^m = SelecioneInstância (A_k, a, $\prod$);
    enquanto ($v_i^m \neq$ nenhum) faça {
        escalone v_i^m em s_k;
        $INST_k = INST_k \cup \{ v_i^m \}$;
        retire v_i^m de A_k;
        retire v_i^m de I;
        v_i^m = SelecioneInstância (A_k, a, $\prod$);
    retorne($INST_k$)

Algoritmo 4.6: Método para escalonamento de um estado.

SelecioneInstância (A_k, a, $\prod$)
escolha a instância $v_i^m \in A_k$ com maior prioridade $\prod$ que satisfaz a restrição a;
retorne (v_i^m);

Algoritmo 4.7: Método para selecionar instância sob restrição de recursos.

O algoritmo implementado requer a inserção de elementos em A_k e a remoção do elemento com maior prioridade, procedimentos que são implementados, respectivamente, pelos seguintes métodos:

- Insira (v_i^m, A_k): insere a instância v_i^m no conjunto A_k;
- RemovaMax(A_k): remove e retorna o elemento de A_k com a maior prioridade.

Como tais procedimentos são muito freqüentes no escalonamento, escolhemos uma estrutura de dados do tipo "Binary Heap" para implementar eficientemente o conjunto A_k como uma fila de prioridades. Uma fila de prioridades é uma estrutura de dados para manter um conjunto de elementos cada qual associado com um valor chamado *chave de ordenamento*. Em nossa implementação a chave de um elemento corresponde à sua posição em uma dada codificação de prioridade $\prod$, ou seja, se $\prod$ (j) = v_i^m, então j é a chave de v_i^m na "Binary Heap".

Em uma "Binary Heap" o tempo de execução de um método de inserção ou remoção é da ordem de O(log n), onde n é o número de elementos armazenados. Assim, através da escolha deliberada de uma "Binary Heap" para implementar o conjunto A_k, estamos preparando o terreno para suportar eficientemente grafos com um grande número de vértices.

4.3 Experimentos e Resultados

Nos experimentos foram utilizados dois exemplos clássicos, extraídos da literatura de Síntese de Alto Nível. O exemplo FDCT é um algoritmo que calcula a "fast discrete cosine transform" e foi retirado de [MALL90]. O exemplo WDELF é um algoritmo que implementa um filtro de onda digital de quinta ordem e foi retirado de [DEWI85]. A

Tabela 4.1 resume as principais características dos DFG's resultantes dos exemplos adotados neste trabalho para fins de "benchmarking".

Tabela 4.1: Resumo das características dos exemplos usados como benchmarks

Exemplo	Vértices	Arestas	Tipo de operações
FDCT	44	68	13 adds, 13 subs e 16 mults
WDELF	36	66	25 adds, 1 sub e 8 mults

Nos números de vértices e arestas da Tabela 4.1, estão computados os pólos dos grafos (fonte e sumidouro), bem como todas as arestas emergentes do vértice fonte e as incidentes no vértice sumidouro. Tais vértices e arestas estão associados a atrasos de execução nulos, não interferindo no processo de escalonamento utilizando paralelização de laços.

Os experimentos aqui descritos foram realizados sob a hipótese de que o atraso de execução de todas as operações é unitário, exceto as multiplicações, cujo atraso de execução é igual a dois ciclos de relógio.

As operações de adição e subtração podem ser executadas em um mesmo operador, denominado de unidade lógico-aritmética (abreviadamente ALU). Assim, diferentes tipos de operações podem ocupar o mesmo operador (em diferentes ciclos de relógio). Já as operações de multiplicação podem ocupar apenas o operador denominado multiplicador (abreviadamente MULT).

Foram pesquisadas em nossos experimentos 1000 soluções alternativas para os dois exemplos utilizados.

4.3.1 Os resultados do escalonamento sem paralelização

A Tabela 4.2 compara o dii obtido de nossa abordagem com os resultados reportados em [HEIJ96] para o exemplo FDCT. As duas primeiras colunas desta tabela representam as restrições de recursos. A terceira coluna mostra o menor dii que pode ser

encontrado para uma dada restrição de recursos. A quarta coluna contém os resultados reportados na literatura [HEIJ96] utilizando um algoritmo de "List Scheduling". As quinta e sexta colunas representam a melhor solução obtida em nossa abordagem ao se minimizar o dii.

Tabela 4.2: dii para o exemplo FDCT(sem paralelização).

Restrição de Recursos		[HEIJ96]		Nossa Abordagem	
MUL	**ALU**	**Ótimo dii**	**List Scheduling dii**	**dii**	**λ**
8	4	8	8	8	8
5	5	10	10	10	10
4	3	11	13	11	11
4	2	13	15	13	13
3	2	14	17	15	15
2	2	18	21	18	18
2	1	26	27	26	26
1	1	34	40	34	34

Note nesta tabela que apenas em um dos casos nossa abordagem não conseguiu encontrar o valor ótimo de dii. Isto se deve a uma limitação do algoritmo "List Scheduling" (LS), adotado em nosso escalonador. Tal limitação é ilustrada na Figura 4.1, que foi retirada de [HEIJ96]. A Figura 4.1a mostra parte de um DFG, enquanto as Figuras 4.1b e 4.1c ilustram duas soluções distintas sob uma mesma restrição de recursos: 1 MULT e 1 ALU. A primeira é a solução gerada pelo algoritmo LS, cuja heurística é a de escalonar todas as instâncias que puderem ocupar os recursos disponíveis. A segunda é a solução ótima. Suponha que as operações que mapeiam para o multiplicador levam dois ciclos para executar e as que mapeiam para a unidade lógico-aritméticas levam um ciclo. Observe que, como o MULT está livre, o algoritmo LS escalona a operação "b" no primeiro ciclo, independentemente da prioridade Π. Com isso gera uma solução com λ=6. Por outro lado, o adotado por [HEIJ96], denominado de "Topological Permutation Scheduling"(TPS), ao adiar a ocupação de um recurso (MULT), consegue gerar uma solução com λ=5.

Como o TPS (utilizado em [HEIJ96]) é mais geral do que o LS (utilizado em nosso escalonador), ele pode construir soluções que nossa abordagem é incapaz de gerar. Isso explica o porquê não obtemos melhor dii em todos os casos.

Uma outra deficiência da heurística do algoritmo LS será discutida na Seção 4.4.3.

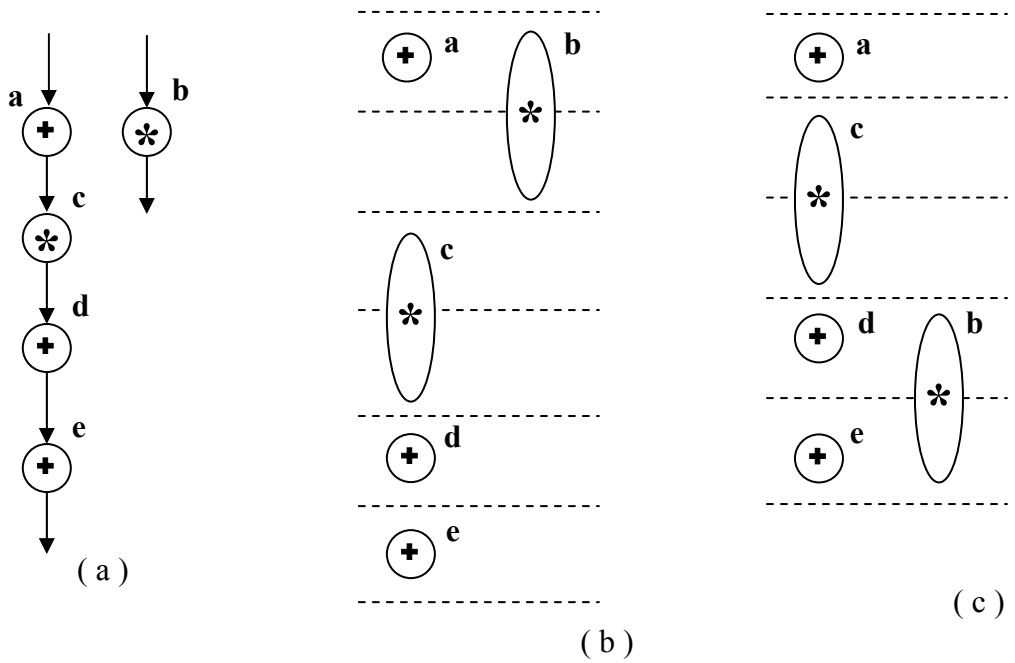


Figura 4.1: Exemplo da limitação do "List Scheduling" [HEIJ96].

A Tabela 4.3 compara nossos resultados com os reportados em [HEIJ96] para o exemplo WDELF, sem paralelização. Para este exemplo, o custo ótimo foi obtido para todas as restrições de recurso testadas.

Tabela 4.3: dii para o exemplo WDELF (sem paralelização).

Restrição de Recursos		[HEIJ96]		Nossa Abordagem	
MUL	**ALU**	**Ótimo dii**	**List Scheduling dii**	**dii**	**λ**
3	3	17	17	17	17
2	2	18	19	18	18
1	2	21	21	21	21
1	1	28	28	28	28

Observe que trabalhando com W = 1, ou seja, sem paralelização, nossa abordagem obteve os valores ótimos para todas as restrições de recursos testadas, exceto para um dos casos. Isto demonstra que o algoritmo funciona correta e eficientemente, quando o paralelismo é restrito a uma única iteração.

4.3.2 Impacto da paralelização no dii

A Tabela 4.4 apresenta uma comparação entre os resultados obtidos em nossa abordagem com os reportados na literatura. A duas primeiras colunas da tabela representam as restrições de recursos. A terceira e a quarta coluna mostram o dii e a latência, respectivamente, conforme relatados em [FRAM94]. De forma similar, a quinta e sexta colunas apresentam resultados reportados em [RADI96]. As nove últimas colunas relatam o dii e a latência encontrados em nossa abordagem para diferentes tamanhos de janelas (W). Note que para cada W temos dois valores de latências representadas: λ é a latência do laço (vide Definição 2.14) e λ' é a máxima latência do conjunto de iterações que começam e terminam dentro do "kernel" (excluindo, portanto, as iterações que começam no "pre-amble". É importante frisar que os resultados relatados em nossa abordagem correspondem às soluções com menor dii obtidas para cada restrição de recursos.

Tabela 4.4: Comparação do dii para WDELF (com paralelização).

Restrição de Recursos		[FRAM94]		[RADI96]		Nossa abordagem W=2			W=3			W=4		
MULT	**ALU**	**dii**	**λ**	**dii**	**λ**	**dii**	**λ**	**λ'**	**dii**	**λ**	**λ'**	**dii**	**λ**	**λ'**
3	4	19	16	-	-	11	21	17	7	24	20	7	31	25
3	3	-	-	16	18	10	22	18	9	30	25	9	41	34
2	3	17	17	16	18	11	24	21	9	32	27	9	43	33
2	2	18	18	17	19	13	29	22	13	42	36	13	57	50
1	2	20	20	19	21	16	37	31	16	55	47	16	72	63
1	1	29	28	-	-	26	49	37	26	78	64	26	109	97

Observe que nossa abordagem obteve melhores valores de dii em todos os casos. Por exemplo, para a restrição de recursos de 2 multiplicadores e 3 unidades lógico-aritméticas, obteve-se valores para dii de 11 (W=2) e 9 (W=3 e W=4), enquanto o melhor dii reportado na literatura foi de 16 [RADI96]. Note, porém, que enquanto a

latência relatada na literatura foi de 17, nossa abordagem obteve λ=24 (W=2), 32 (W=3) e 43 (W=4). Portanto, aparentemente, nossa abordagem obteve um menor dii às custas de uma maior latência. Observe também que, para este exemplo, $\lambda' < \lambda$ em todos os casos.

A melhor qualidade dos valores de dii resultantes de nossa abordagem poderia, à primeira vista, despertar dúvida quanto à efetiva correspondência entre o "benchmark" utilizado em [FRAM94] e [RADI96] e aquele utilizado em nossa abordagem. Entretanto, a hipótese de que nossos melhores resultados pudessem advir de uma especificação incorreta do "benchmark" é afastada, pois os valores obtidos com W=1 conferem com os retirados da literatura, conforme mostrado nas Tabelas 4.2 e 4.3.

Além disso, são fortes as evidências de que a implementação do algoritmo não contém erros, pois as melhores soluções obtidas foram conferidas automaticamente e inspecionadas manualmente. Foram executados testes para averiguar se as restrições de precedência e as restrições de recursos foram respeitadas a cada estado. Exemplos de que as soluções obtidas foram construídas corretamente podem ser encontrados nos Anexos A e B, que apresentam um escalonamento gerado via "loop pipelining" e outro gerado via "loop unrolling".

Tabela 4.5: Comparação do dii para WDELF (resultados em [HEIJ96]).

	Restrição de Recursos		**[HEIJ96]**		**Nossa Abordagem (W=2)**		
	MULT	**ALU**	**dii**	**λ**	**dii**	**λ**	**λ'**
A	3	4	16	14	11	17	17
B	3	3	16	14	10	16	14
C	2	3	16	17	11	16	16
D	2	2	17	15	13	18	15
E	1	2	19	21	16	17	17
F	1	1	28	28	26	24	16

A Tabela 4.5 mostra uma comparação dos resultados reportados em [HEIJ96] com os obtidos em nossa abordagem para o exemplo WDELF. Embora esse autor não deixe explícito no texto qual a sua métrica de latência, ele provavelmente utiliza uma forma

distinta da utilizada nesta dissertação. A distinção entre essas métricas podem ser melhor entendida examinando-se a discussão no Anexo C.

Para viabilizar a comparação de nossos resultados com os reportados em [HEIJ96], a provável métrica utilizada por aquele autor foi adotada – provisoriamente – em nossa abordagem, para o experimento especifico relatado na tabela da página anterior.

A Tabela 4.5 mostra a comparação para os casos de restrição de recursos enumerados de A a F. Note nessa tabela que os valores de dii encontrados em nossa abordagem são sempre menores do que os reportados em [HEIJ96].

4.3.3 Impacto da paralelização na latência

Esta seção relata experimentos de minimização de latência. Encontrou-se somente um autor que relatou resultados em experimentos similares [HEIJ96]. Conforme citado na seção anterior, esse autor utiliza uma métrica distinta da utilizada nesta dissertação. Por isso, novamente adotamos provisoriamente a métrica utilizada por [HEIJ96] (vide Anexo C) para as comparações da Tabela 4.6.

Tabela 4.6: Comparação da latência para WDELF (com paralelização).

	Restrição de Recursos		**[HEIJ96]**		**Nossa Abordagem (W = 2)**	
	MULT	**ALU**	**dii**	**λ**	**dii**	**λ**
A	3	4	16	14	15	14
B	3	3	16	14	18	14
C	2	3	16	17	22	14
D	2	2	17	15	24	15
E	1	2	19	21	35	15
F	1	1	28	28	49	18

Note que, em todos os casos testados, encontraram-se valores de latência iguais ou melhores que os reportados em [HEIJ96]. Todavia, em apenas um caso (A), encontrou-se um melhor dii.

Para complementar a comparação da Tabela 4.6, em que nossas melhores latências foram relatadas independentemente dos valores de dii, um novo experimento foi realizado de forma que as latências pudessem ser comparadas para o mesmo valor de dii. Os resultados são mostrados na Tabela 4.7.

Tabela 4.7: Comparação da latência para os mesmos dii em WDELF.

	Restrições			[HEIJ96]	Nossa Abordagem (W = 2)
	MULT	**ALU**	**dii**	**λ**	**λ**
A	3	4	16	14	14
B	3	3	16	14	15
C	2	3	16	17	17
D	2	2	17	15	17
E	1	2	19	21	20
F	1	1	28	28	29

Note que para o caso E encontrou-se uma latência menor que a reportada na literatura enquanto que, para os casos A e C, um mesmo valor de latência foi encontrado. Note também que para os demais casos (B, D e F) foram encontrados valores superiores aos relatados na literatura. É provável que não tenha sido possível encontrar os mesmos valores devido a mesma deficiência do escalonador descrita ao final da Seção 4.3.2.

4.3.4 Impacto no espaço de soluções

O Gráfico 4.1 representa o espaço de soluções para o WDELF, sob uma mesma restrição de recursos (6 MULTs e 6 ALUs). O espaço de soluções é representado em termos de dii e latência, para diferentes valores de W.

Note que para W = 2 tem-se um espaço de solução restrito e que este espaço aumenta à medida que aumentamos W. Isto ocorre pelo fato de expormos paralelismo entre mais instâncias. Note também que quando aumentamos W, o espaço de soluções alcança melhores valores de dii. Entretanto, o aumento de W tende a aumentar as latências. Isto reforça o fato de que melhores valores de dii foram encontrados às custas de uma pior latência (Seção 4.3.2). Em resumo, analisando o espaço de soluções para uma mesma restrição de recursos e diferentes W, pode-se concluir que os melhores valores de dii

estão diretamente relacionados com o aumento da latência e com o aumento do espaço de soluções.

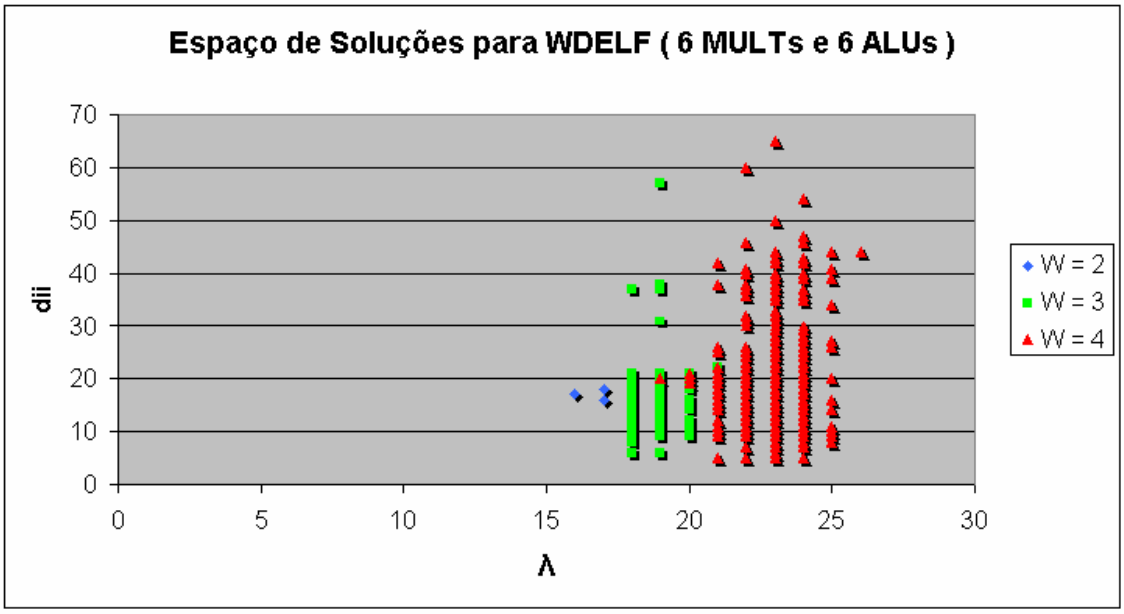


Gráfico 4.1: Espaço de soluções para WDELF (6 MULTs e 6 ALUs).

O Gráfico 4.2, também representa o espaço de soluções para o exemplo WDELF. Porém, este gráfico mostra os espaços de soluções obtidos para diferentes restrições de recursos, mas para uma janela de tamanho fixo (W=2).

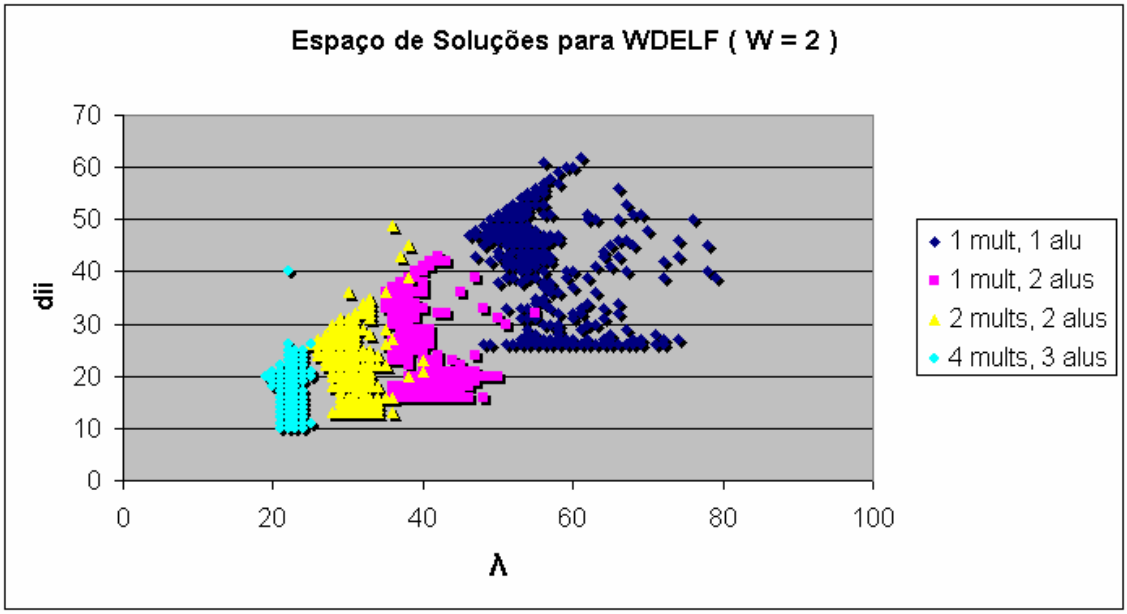


Gráfico 4.2: Espaço de soluções para WDELF (W = 2).

Note que o espaço de soluções converge para valores cada vez melhores (tendendo para dii = 0 e latência = 0) à medida que aumentamos o número de recursos. Isso se deve ao fato de que à medida que aumentamos o número de recursos, as instâncias podem ser neles melhor acomodadas resultando em menor latência e dii.

4.3.5 Análise “loop unrolling” x “loop pipelining”

O teste de equivalência da Definição 2.19 é condição necessária para se obter "loop pipelining", mas não é condição suficiente. Como conseqüência, são geradas soluções onde a equivalência de estados resulta em “loop unrolling” [HENN96].

O Gráfico 4.3 mostra a proporção de soluções encontradas através de “loop unrolling” e de “loop pipelining”. A altura das barras empilhadas representa a porcentagem de soluções resultantes de paralelização via “loop unrolling” e “loop pipelining”. Cada barra vertical está associada a diferentes restrições de recursos enumeradas de A a S, de forma que o número de recursos aumenta de A para S, conforme pode ser visto na Tabela 4.8.

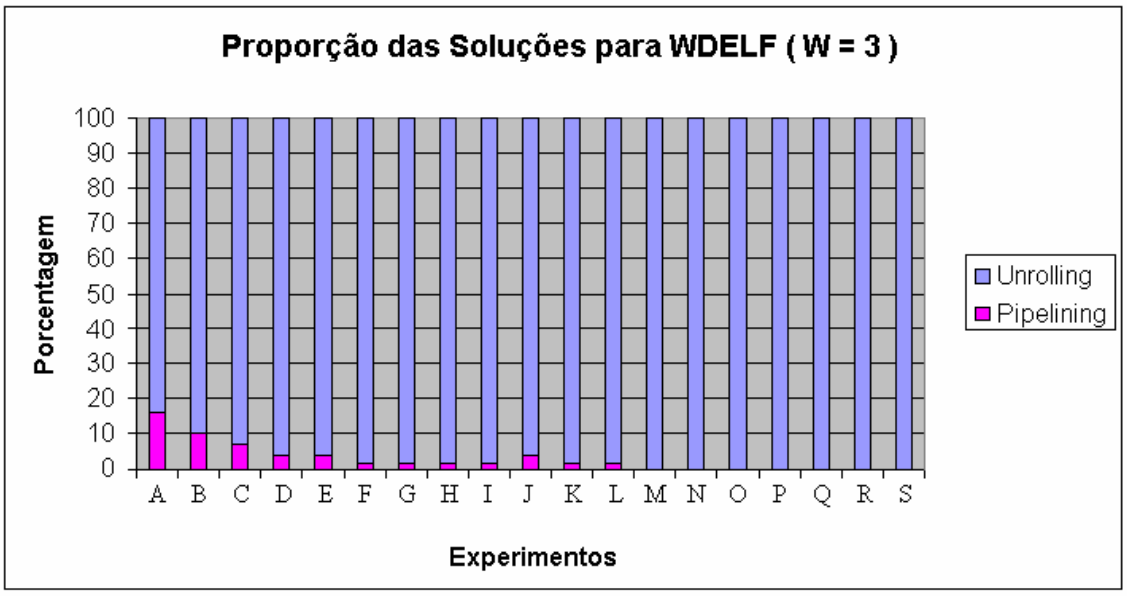


Gráfico 4.3: Porcentagem de soluções geradas por “unrolling” e “pipelining”.

Note primeiramente que há uma grande diferença nas proporções entre as soluções geradas via "loop pipelining" e "loop unrolling". Note também que, para os casos onde existe maior abundância de recursos (M até S), nenhuma das soluções encontradas foi gerada via "loop pipelining". Isto se deve a uma limitação do escalonador, que será discutido na Seção 4.4.3.

Tabela 4.8: Casos de restrição de recursos

	Casos																		
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**
MULTs	1	1	2	2	3	4	4	3	4	5	5	6	7	6	8	10	12	14	16
ALUs	1	2	2	3	3	3	4	4	5	5	6	6	6	7	8	10	12	14	16

4.4 Limitações do Protótipo

Para implementação do algoritmo de "loop pipelining", algumas limitações para tornar viável o estudo sobre o mesmo foram adotadas. São estas limitações:

- O algoritmo atual supõe que o laço seja infinito, ou seja, a condição de parada do laço não é testada. O algoritmo aplica o escalonamento considerando o laço infinito e termina sua execução sempre que um estado equivalente é encontrado;
- Para uma maior simplificação, assume-se também que o corpo do laço esteja isento de construções condicionais.

Tais limitações são abordadas em trabalhos futuros (veja Capítulo 5).

4.4.1 Limitação no tratamento da especificação de entrada

Por simplicidade, assumiu-se neste trabalho somente dependência de dados entre instâncias de operações de uma mesma iteração. No entanto, alguns laços possuem dependências entre instâncias de operações de diferentes iterações e são denominadas de "loop-carried dependences" [HENN96]. A Figura 4.2 mostra um exemplo de um algoritmo onde ocorre esse tipo de dependência.

Neste exemplo, o número da iteração atual é dado pelo contador i. Observe que, como os valores produzidos e consumidos são armazenados em arranjos, o índice de um arranjo indica o número da iteração onde o valor do elemento indexado foi produzido. Por exemplo, note que a execução do primeiro comando dentro do laço depende de valores produzidos por instâncias de operações executadas nas duas iterações anteriores, ou seja, a elemento de "a" na iteração atual (i) recebe o elemento de "b" da iteração i-1 somado ao elemento de "c" da iteração i-2.

```
for ( i=1; i=i+1; i<=10)
{
      a[i] = b[i-1] + c[i-2]
      b[i] = a[i] + 1;
      c[i] = a[i] * b[i];
}
```

Figura 4.2: Exemplo de laço contendo "loop-carried dependences".

Esse tipo de dependência não foi modelado nem implementado, mas será objeto de trabalho futuro (veja Capítulo 5).

4.4.2 Limitação do explorador

O Gráfico 4.4 ilustra a distribuição do dii desde a primeira até a centésima solução pesquisada para o exemplo WDELF. Note que, apesar da geração de soluções alternativas permitir a exploração do espaço de projeto, evidencia-se aqui, que o processo de busca não mostra uma convergência nítida para soluções cada vez melhores. Isto é uma conseqüência da limitação atual da implementação do explorador. A remoção desta limitação será objeto de um trabalho complementar de pesquisa no âmbito do projeto OASIS, conforme será explicado no Capítulo 5.

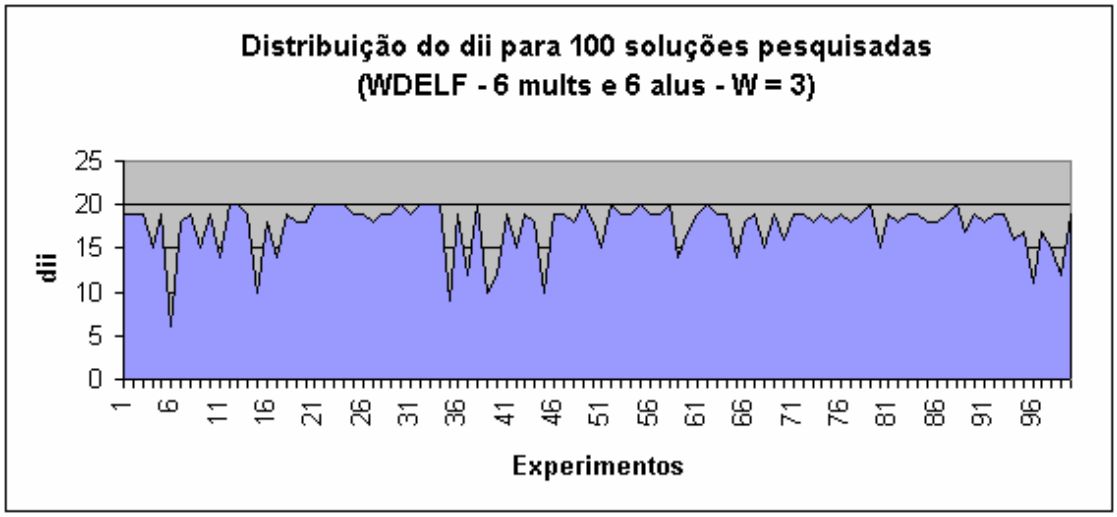


Gráfico 4.4: Distribuição do dii para 100 soluções pesquisadas (WDELF).

4.4.3 Limitação do escalonador

O Gráfico 4.5 mostra o dii e a latência encontrados para o escalonamento de WDELF sob uma restrição de 6 MULTs e 6 ALUs para W = 2. A distribuição foi feita para diversas restrições de recursos (de A até S) conforme já descrito na Tabela 4.8.

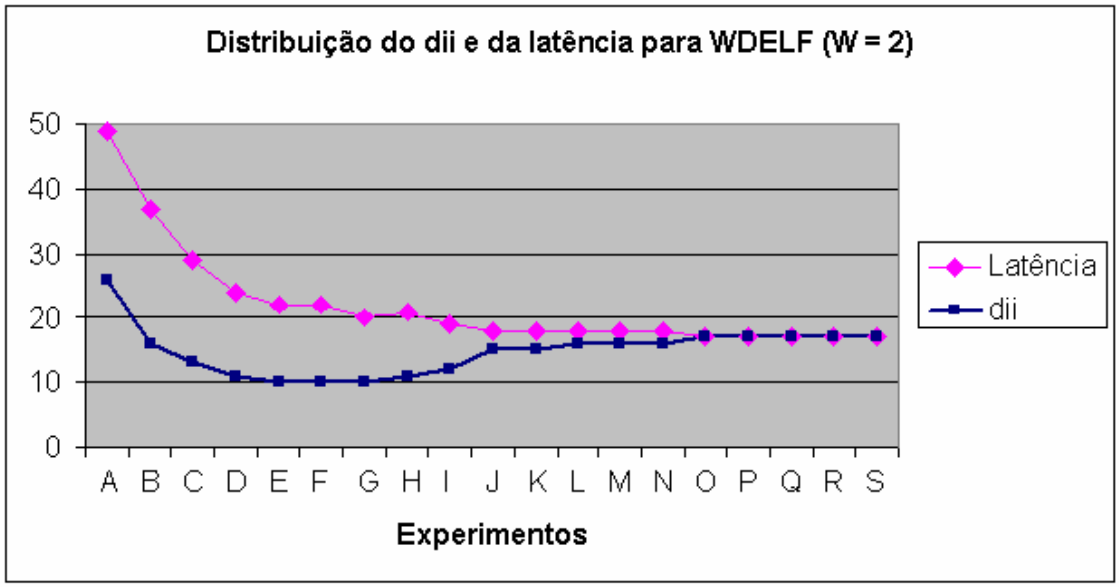


Gráfico 4.5: Distribuição do dii e da latência para WDELF (W = 2).

Note que para os casos de A à G, ao se aumentar os recursos, obteve-se uma melhora no dii e na latência, ou seja, melhores soluções foram encontradas com o aumento dos recursos. Entretanto, para os casos de H a S obteve-se piores resultados de dii ao se aumentar os recursos.Um comportamento similar pode ser observado no Gráfico 4.6, que é em tudo análogo ao anterior, mas apresenta resultados obtidos para W=3.

Observe que o ponto de inflexão da curva a partir do qual os valores de dii começam a piorar é distinto nos Gráficos 4.5 e 4.6. No Gráfico 4.5, tal ponto de inflexão corresponde à restrição G; no Gráfico 4.6, à restrição L. Isto se deve a diferentes balanceamentos entre o paralelismo exposto e o paralelismo acomodável nos recursos disponíveis. No Gráfico 4.5 o paralelismo exposto corresponde a instancias de duas iterações (W=2), o qual é totalmente acomodado nos recursos correspondentes à restrição G. Portanto, um aumento de recursos deveria produzir resultados melhores, uma vez que todo o paralelismo exposto já estaria acomodado. No Gráfico 4.6, o balanceamento do paralelismo ocorre para um maior número de recursos, correspondente à restrição L, porque mais recursos são necessários para acomodar o maior paralelismo exposto, que agora corresponde a instâncias de três iterações (W=3).

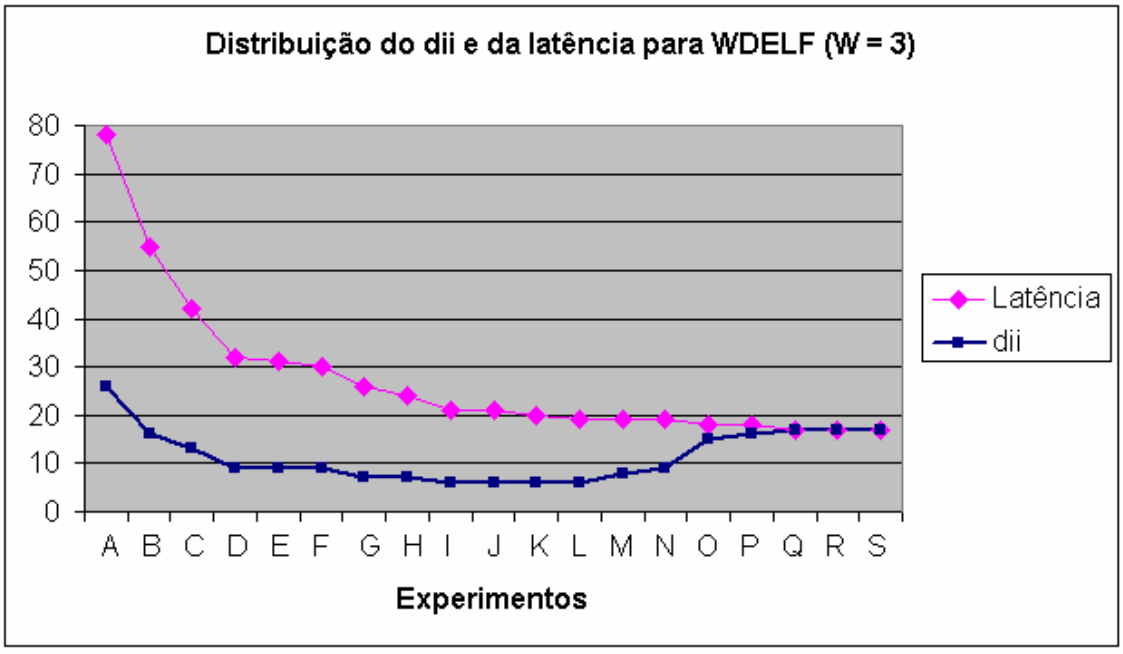


Gráfico 4.6: Distribuição do dii e da latência para WDELF (W = 3).

Todavia, o balanceamento por si só não explica o comportamento anômalo que leva à degradação do dii. Esse comportamento pode ser atribuído a uma deficiência do algoritmo de escalonamento utilizado no escalonador, ou seja, o "List Scheduling" (LS). Conforme explicado na Seção 4.3.5, a heurística do LS procura escalonar o maior número de instâncias possíveis em um estado, enquanto existirem recursos disponíveis. Isso faz com que o dii torne-se cada vez mais independente de Π, conforme os recursos vão se tornando mais abundantes. Essa independência ocorre porque quando existem recursos livres em número suficiente, todas as instâncias disponíveis são escalonadas, qualquer que seja a ordem delas em Π. Dessa forma, várias codificações de prioridade distintas resultam em escalonamentos idênticos. Felizmente, os projetos típicos têm restrições severas de recursos, fazendo com que essa deficiência tenda a não ser observada em casos práticos.

Capítulo

5 Conclusões e Trabalho futuro

Esta dissertação apresentou uma abordagem que permite a exploração de soluções alternativas para um problema clássico de Síntese de Alto Nível: o escalonamento com restrição de recursos. Como resultado da modelagem e da implementação apresentadas neste trabalho, algumas contribuições podem ser apontadas:

- **Escolha e implementação de um algoritmo de Paralelização:** fez-se a escolha de um algoritmo de paralelização, procurando-se encontrar um algoritmo geral o suficiente e que permitisse adaptações, possibilitando o futuro tratamento de construções condicionais e de "loop-carried dependences";
- **Proposta de um novo critério de equivalência de estados:** no Algoritmo de Aiken [AIKE95], dado um estado, os recursos que se encontram disponíveis para acomodar instâncias naquele estado são representados por uma *tabela de reserva* R_k. Ao contrário do algoritmo original, que utiliza A_k e R_k para encontrar estados equivalentes, nossa abordagem utiliza A_k e U_k. Essa modificação proporcionou duas vantagens:
  a) Independência da modelagem de ocupação de recursos: A modelagem de ocupação de recursos pode prescindir da tabela de reserva R_k requerida em [AIKE95]. Assim, poder-se-á adotar durante o escalonamento, uma modelagem alternativa de restrição de recursos como a proposta em [MESM97], a qual tem a vantagem de unificar o tratamento de todas as restrições (precedência, tempo e recurso) modelando-as através de arestas.
  b) Manutenção eficiente da lista U_k: Uma outra contribuição que pode ser mencionada é a implementação desta lista como uma "Binary Heap", onde o tempo de execução para a inserção e remoção de elementos é da ordem O(log n). Isso nos dá suporte para realizarmos experimentos com grafos com um grande número de vértices de maneira mais eficiente e rápida.

- **Análise do espaço de soluções em termos de dii x latência:** O foco deste trabalho foi avaliar o impacto de paralelização de laços no dii, mas também se fez uma avaliação do impacto na latência. Com esses experimentos, fez-se uma análise do espaço de soluções em termos de dii e latência, para diferentes restrições de recursos e diferentes graus de paralelismo exposto.
- **Análise da proporção de soluções geradas via "pipelining" x "unrolling":** Conforme relatado em nosso trabalho anterior [FERR02], acreditava-se que as soluções geradas via "loop unrolling" conduzissem a resultados inferiores se comparadas com os das soluções que geradas via "loop pipelining". No entanto, analisando a proporção entre elas, obtivemos soluções onde o número de soluções geradas via "loop unrolling" foi muito maior que o número de soluções geradas via "loop pipelining", sendo também significativas, pois em alguns casos, encontrou-se a melhor solução entre as soluções geradas via "loop unrolling".
- **Obtenção de melhores intervalos de introdução de dados:** Encontrou-se, através de nossa abordagem, soluções com intervalos de introdução de dados menores do que os que foram encontrados na literatura, porém às custas de um aumento na latência.

Outras contribuições serão feitas no âmbito do projeto OASIS, mas fora do âmbito dessa dissertação, como:

- **Suporte a "loop-carried dependences"**: Como serão necessários novos atributos para representar tais dependências, é necessário fazer um estudo detalhado de trabalhos correlatos para obter uma modelagem compatível com a deste trabalho;
- **Suporte a laços onde o corpo contenha condicionais**: Esse tópico não foi incorporado à dissertação de mestrado por ser de complexidade incompatível com a atual limitação de tempo para a conclusão da dissertação. Como o tratamento de construções condicionais foi objeto de trabalho recente no âmbito do projeto OASIS, ele poderá ser combinado com "loop pipelining" em um futuro trabalho de pesquisa.
- **Extensão da inteligência do explorador**: É possível melhorar a obtenção de diferentes codificações de prioridade com algoritmos que usem critérios

evolucionários para ordenar as codificações Π, na busca de recodificações que levem a soluções cada vez melhores. Este tópico não foi incorporado à dissertação, pois necessita do conhecimento de técnicas de natureza bastante diferente das até aqui estudadas e implementadas. Isto não traz grandes prejuízos, pois este tópico pode ser abordado futuramente de forma totalmente ortogonal a este trabalho.

- **Filtragem do espaço de soluções evitando soluções inferiores**: É possível evitar a construção de soluções inferiores aplicando restrições de tempo *durante* o escalonamento. Isso equivale a filtrar o espaço de solução, fazendo com que somente soluções de melhor qualidade sejam efetivamente pesquisadas. Isso pode ser feito utilizando o Algoritmo de Bellman-Ford combinado com técnicas de análise de restrições de tempo [MESM97] . Esse tópico é objeto de estudo de um dos membros do projeto OASIS.

Referências Bibliográficas

[AIKE95] AIKEN, A.; NICOLAU A., Resource-Constrained Software Pipelining, IEEE Transactions on Parallel and Distributed Systems, vol. 6, nº 12, Dezembro 1995.

[AIKE88] AIKEN, A.; NICOLAU A., Perfect pipelining: A new loop parallelization technique. Proc. European Symposium on Programming, pp. 221-235, 1988.

[CAMP91] CAMPOSANO, R., Path-based scheduling for synthesis, IEEE Trans. On Computer-Aided Design, vol 10 nº 1, Janeiro 1991.

[DEMI94] DE MICHELI, GIOVANNI, Synthesis and Optimization on Digital Circuits, Mc Graw-Hill, 1994.

[DEWI85] DEWILDE et. al., Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms, in S.Y. Kung, H. J. Whitehouse and T. Kailath, VLSI and Modern Signal Processing, Prentice Hall, 1985.

[EBCI87] K. EBCIOGLU, A compilation technique for software pipelining of loops with conditional jumps, Proc. 20th Annual Workshop on Microprogramming, pp 69-79, Dezembro 1987.

[FERR02] FERRARI, DIONE JONATHAN, Paralelização de Laços: Um Estudo Introdutório, UFSC, Março 2002.

[FRAN94] F. FRANSEN, Retiming for Dataflow Graphs, Training Report, Eindhoven University of Technology, October 94.

[FROL96] FRÖLICH, M. WERNER, DaVinci V2.0x Online Documentation, Universität Bremen, Germany, June 1996.
(http://www.tzi.de/~davinci/doc_V2.0)

[HEIJ96] HEIJLIGERS, M. J. M., The Application of Genetic Algorithms to High-Level Synthesis, PhDS. Thesis, Eindhoven University of Technology, The Netherlands, 1996.

[HENN96] HENNESSY, J. L. and PATTERSON, D. A., Computer Architecture – A Quantitative Approach, 2nd edition, Morgam Keufmann Publisher Inc., 1996.

[KDE01] KDE.com, 2001, disponível em www.kde.com, acessado em 15 de Julho de 2001.

[MALL90] MALLON, D. J.; DENYER, P. B., A New Approach to Pipeline Optimization, Proc. EDAC'90m oo, 1990.

[MESM97] MESMMAN, BART et. al., Constraint Analysis for DSP Code Generation, 10th International Symposium on Synthesis Antwerp, Belgium, 1997.

[MLAN88] M. LAM, Software pipelining: An effective scheduling technique for VLIW machines, Proc. SIGPLAN'88 Conf. On Programming Languages Design and Implementation, pp. 318-328, Junho 1988.

[RADI96] RADIVOJEVIC, I.; BREWER, F., A New Symbolic Technique for Control Dependent Scheduling, IEEE Transactions on Computer-Aided Design, vol. 15, n. 1, 1996.

[SANT00] SANTOS, LUIZ C. V. DOS, A Síntese de Alto Nível na Automação de Projeto de Sistemas Computacionais, Capítulo 8 do livro-texto da VIII Escola de Informática da SBC-Sul, maio de 2000, pp. 211-231.

[SANT98] SANTOS, LUIZ C. V. DOS, Exploiting Instruction-level Parallelism: A Constructive Approach, Eindhoven University of Technology, PhD. Thesis, 1998.

[WEIS96] WEISS, MARK ALLEN, Algorithms, Data Structures, and Problem Solving with C++, USA, 1996.

Anexo

A Resultado experimental gerado via “loop pipelining”

Este anexo apresenta e explica um resultado ilustrativo da melhor qualidade de nossos resultados, já relatado na Tabela 4.4. O resultado aqui descrito refere-se ao escalonamento gerado via “loop pipelining” do exemplo WDELF com uma janela de 3 iterações (W=3), obtido sob uma restrição de recursos de 2 multiplicadores e 3 unidades lógico-aritméticas.

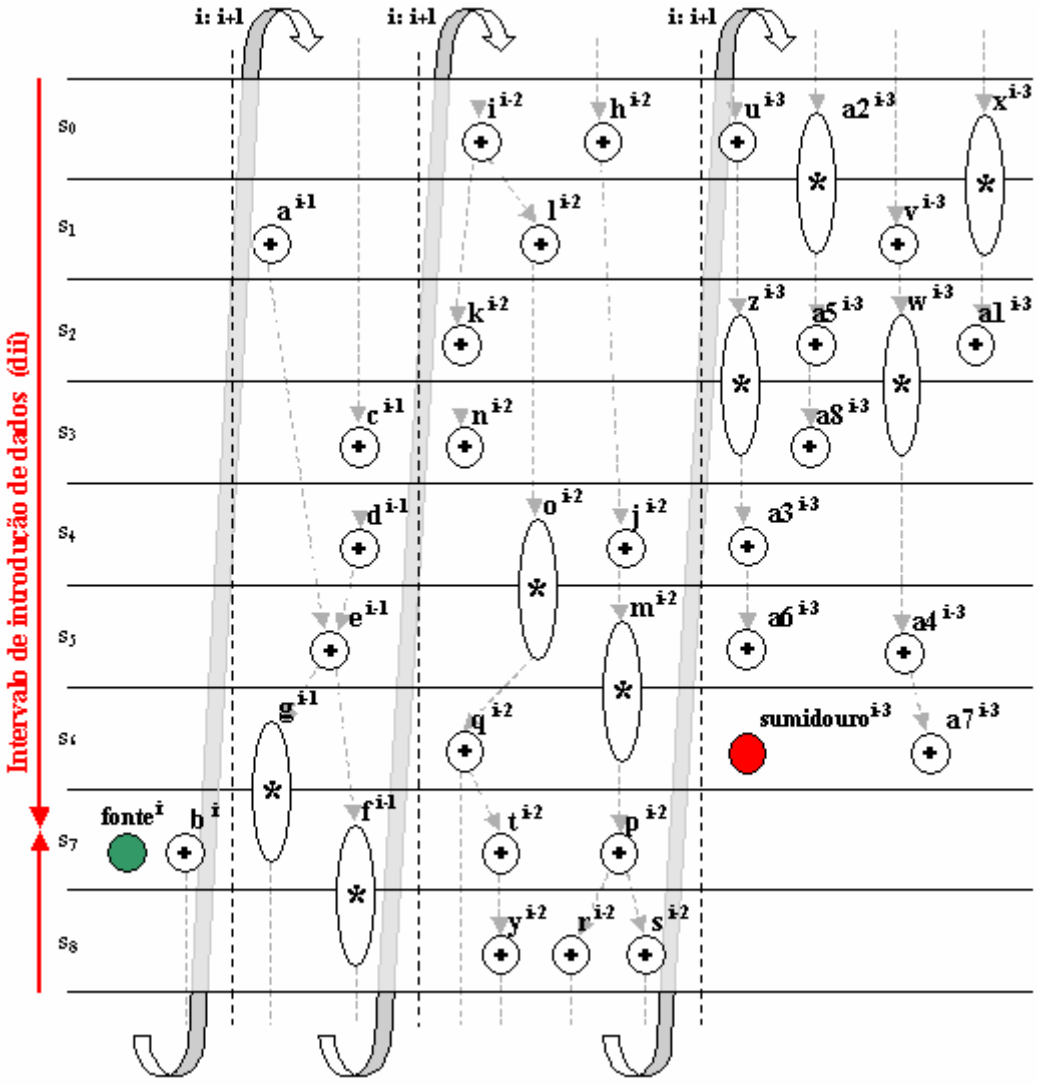


Figura A.1: Resultado experimental ilustrativo de nossa abordagem (“pipelining”).

A Figura A.1, mostra o corpo do laço paralelizado encontrado. Nesta figura, as linhas horizontais separam os estados e, conseqüentemente, mostram em qual estado cada instância da operação é executada. Note que as instâncias que mapeiam para os multiplicadores estão em dois estados, já que elas necessitam de 2 ciclos para executarem. As linhas verticais pontilhadas delimitam as iterações no corpo do laço (representada na figura como iterações i, i-1, i-2, i-3 e i-4) enquanto as linhas horizontais delimitam os estados (representados na figura como os subíndices s_0, s_1, s_2, s_3 e s_4).

Note que, em cada estado, instâncias de diferentes iterações são escalonadas sempre respeitando o tamanho da janela. Por exemplo, no estado s_0 há instâncias das iterações i-1, i-2 e i-3 (W=3). Note também que o número de recursos não é excedido em estado algum.

Observe na figura que as arestas pontilhadas entre as instâncias representam as restrições de precedência e que o escalonamento satisfaz todas essas restrições. Note que para o escalonamento da Figura A.1 tem-se um dii igual a 9. Note também que a execução completa de uma determinada iteração (distância entre fonte e sumidouro através das setas) leva 27 ciclos, ou seja, tem-se uma latência igual a 27.

Anexo

B Resultado experimental gerado via "loop unrolling"

Este anexo apresenta e explica um resultado ilustrativo da melhor qualidade de nossos resultados. O resultado aqui descrito refere-se ao escalonamento gerado via "loop unrolling" do "benchmark" WDELF com uma janela de 3 iterações (W=3), obtido sob uma restrição de recursos de 3 multiplicadores e 4 unidades lógico-aritméticas. Esta solução está relatada na Tabela 4.4.

A Figura B.1, mostra o corpo do laço paralelizado encontrado, que é em tudo análogo ao anterior, mas apresenta um resultado gerado via "loop unrolling". A principal diferença entre eles é que este possui diferentes instâncias da mesma operação dentro do corpo do laço (por exemplo, a instância b^i da operação "b" que foi escalonada no estado s_8 e instância b^{i-1} dessa mesma operação que foi escalonada no estado s_1). Note que a cada iteração do laço o valor de i aumenta em dois e, com isso, o corpo do laço possui dois vértices-fonte e dois vértices-sumidouro.

Observe que, em cada estado, instâncias de diferentes iterações são escalonadas sempre respeitando o tamanho da janela. Por exemplo, no estado s_0 há instâncias das iterações i-2, i-3 e i-4 (W=3). Note também que o número de recursos não é excedido em estado algum.

Observe ainda que para o escalonamento da Figura A.1 tem-se valores distintos de dii, medidos dentro do do corpo do laço (dii1=7 e dii2=6). Como já formalizado em nossas definições, o dii do laço é o maior deles. O mesmo acontece para a latência nesse experimento (λ1=20 e λ2=19).

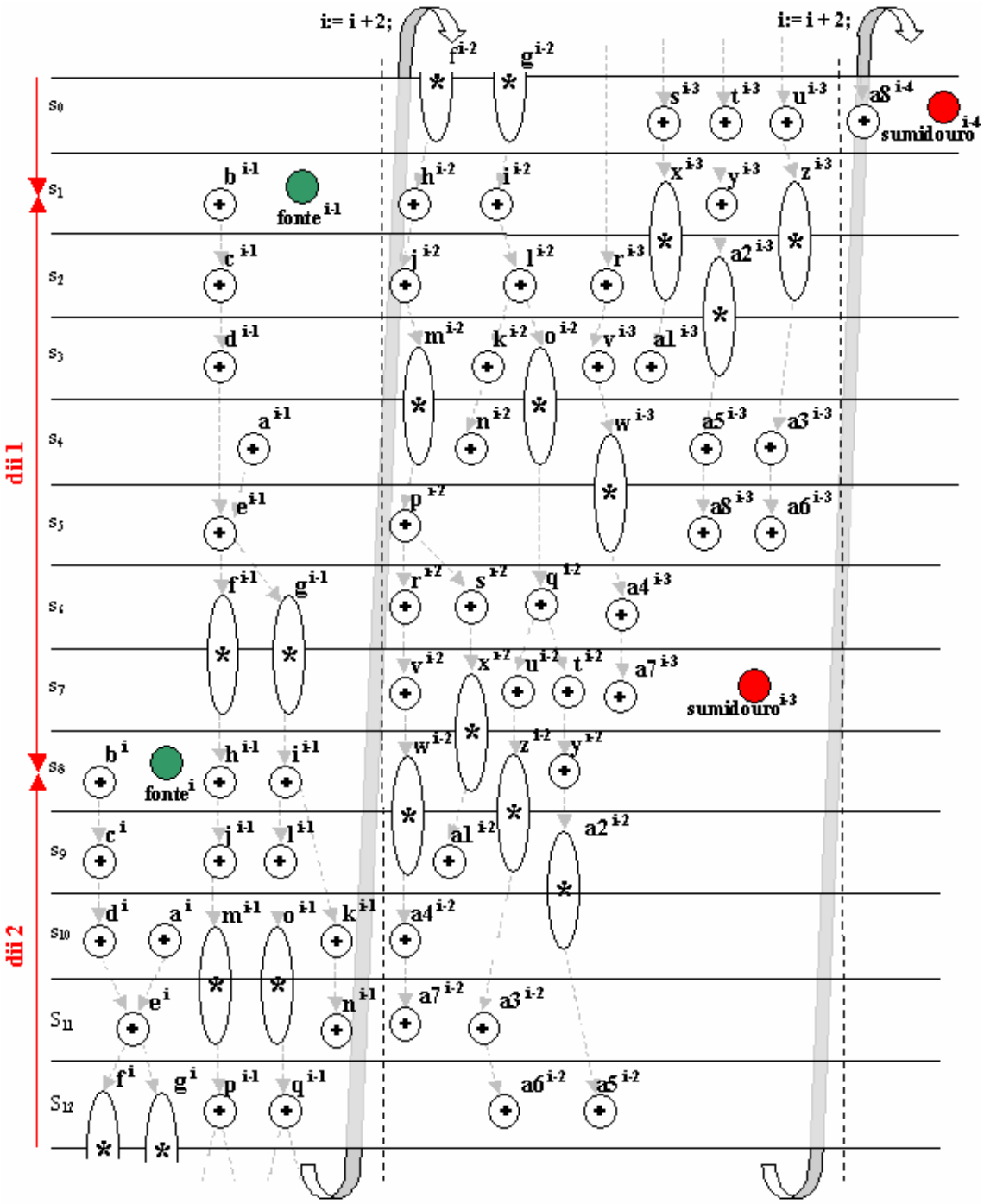


Figura B.1: Resultado experimental ilustrativo de nossa abordagem (“unrolling”).

Anexo

C Métricas de latência

Apresenta-se neste anexo uma interpretação da métrica de latência utilizada em [HEIJ96] e que foi adotada nos resultados experimentais da Seção 4.3.3 deste trabalho. Como esse autor não apresenta de forma explícita essa métrica, fez-se necessária uma interpretação dos resultados por ele reportados através de escalonamentos e descrições dos mesmos.

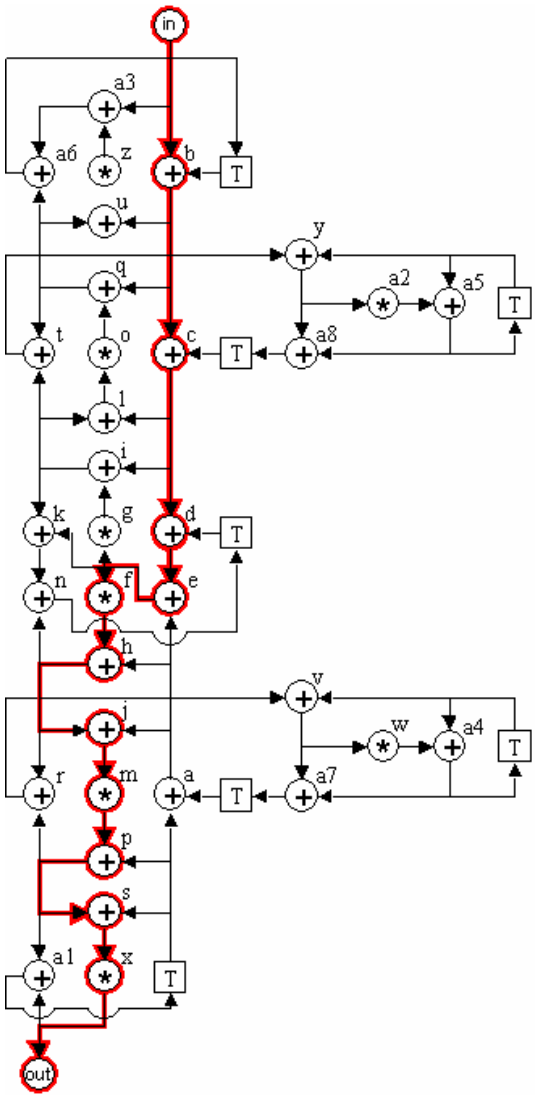


Figura C.1: DFG de um filtro de onda digital de 5ª ordem (WDELF) [HEIJ96].

A Figura C.1 mostra a descrição comportamental do benchmark WDELF (filtro de ondas digitais de 5ª ordem) e foi adaptada de [HEIJ96]. Note que as operações estão representadas pelos círculos (multiplicadores ou somadores) enquanto que as arestas representam a dependência de dados entre essas operações. Note também a existência da fonte (“in”) e do sumidouro (“out”).

De forma distinta da métrica de latência utilizada neste trabalho (veja Definição 2.14), acredita-se que esse autor adotou a latência como sendo a distância entre o estado de chegada dos dados de entradas (“in”) e o estado onde os dados de saída estão disponíveis (“out”) e também que os dados de saída estão disponíveis um estado após o escalonamento da operação “x”. O caminho em destaque na Figura C.1 mostra as operações que ligam “in” e “out”.