Rogério Sigrist Silva

Algoritmos de Síntese de *Pipeline* de Processadores para Sistemas Embutidos: Minimização de Custos, Número de Processadores e Latência

Tese de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientadora: Alice Maria B. H. Tokarnia

Campinas, SP
2006

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

Si38a

Silva, Rogério Sigrist
Algoritmos de síntese de *Pipeline* de processadores para sistemas embutidos: minimização de custos, número de processadores e latência / Rogério Sigrist Silva. --Campinas, SP: [s.n.], 2006.

Orientador: Alice Maria B. H. Tokarnia
Dissertação (Mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Sistemas embutidos de computador. 2. Computadores canalizados. 3. Agenda de execução (administração) – processamento de dados. I. Tokarnia, Alice Maria B. H. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Título em Inglês: Algorithms for processors pipeline synthesis of embedded systems: cost, number of processors and latency minimization
Palavras-chave em Inglês: Embedded system, Scheduling, Pipeline
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Sandra Lucia da Cruz, Ricardo Pannain e Maurício Ferreira Magalhães
Data da defesa: 20/10/2006

Rogério Sigrist Silva

Algoritmos de Síntese de *Pipeline* de Processadores para Sistemas Embutidos: Minimização de Custos, Número de Processadores e Latência

Tese de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Aprovação em: 20/10/2006

Banca Examinadora:

Profa. Dra. Sandra Lucia da Cruz (FEQ/UNICAMP)

Prof. Dr. Ricardo Pannain (IC/UNICAMP)

Prof. Dr. Maurício Ferreira Magalhães (FEEC/UNICAMP)

Campinas, SP

2006

Resumo

Este trabalho descreve três algoritmos para a síntese de sistemas embutidos atendendo à restrição de desempenho representada pela taxa de chegada dos dados, através de uma estrutura de *pipeline* de processadores para execução das tarefas, ao mesmo tempo em que minimizam diferentes parâmetros de qualidade dos sistemas: número de processadores; custo e latência total. Os algoritmos realizam o particionamento *hardware-software* das tarefas, a alocação dos processadores, o mapeamento e escalonamento das tarefas. A alocação de processadores e o mapeamento e escalonamento de tarefas são problemas classificados como NP-Completo e, portanto, foram aplicados métodos heurísticos para suas resoluções.

Como exemplos de aplicação são apresentados os *pipelines* sintetizados pelos algoritmos para grafos sintéticos e para um compressor de áudio digital (AC3). Os *pipelines* sintetizados atingem métricas de qualidade superiores a outros algoritmos publicados.

Palavras-chave: sistemas embutidos, escalonamento, *pipeline*.

Abstract

This work presents three heuristics for synthesizing pipelined embedded systems that satisfy a throughput constraint derived from the maximum input data rate adopting a pipeline structure of processors while minimizing system quality parameters: cost, number of processors, or number of stages. The algorithms perform tasks hardware-software partitioning, processors allocation and task mapping and scheduling. Since processors allocation and task mapping and scheduling are NP-complete problems, heuristic methods were applied.

The examples present the pipelines synthesized by the algorithms for large synthetic systems comparing the quality parameters minimization results and for a real audio compressor (AC3) application. The pipelines reached quality metrics higher than other published algorithms.

Key-words: embedded system, scheduling, pipeline.

Agradecimentos

À minha orientadora, Profa. Alice M. Tokarnia, sou grato pela orientação.
Aos demais colegas de pós-graduação, pelas críticas e sugestões.
Aos profissionais da CPG pela colaboração.
À minha esposa Adriana e a meus filhos Felipe e Guilherme pelo inestimável apoio durante esta longa jornada.

À minha esposa e filhos

Sumário

Sumário vii
Lista de figuras x
Lista de Tabelas xii
Glossário xiii
Capítulo 1 Introdução 1
1.1 Motivação 1
1.2 Objetivo 2
1.3 Pipeline 2
1.3.1 Métricas de desempenho 3
1.4 Organização da tese 5
Capítulo 2 Tecnologia de Processadores 7
2.1 Organização básica de um processador 7
2.2 Processadores genéricos 9
2.3 Processadores dedicados 11
2.4 Processadores específicos para uma área de aplicação 12
2.5 Conjunto de instruções 14
2.5.1 Processadores CISC 14
2.5.2 Processadores RISC 15
2.6 Conclusão 15
Capítulo 3 Projeto de Sistemas Embutidos 17

3.1 Sistemas Embutidos 17
3.2 Projeto de Sistemas Embutidos 18
3.3 Especificação do sistema 20
3.4 Conclusão 25
Capítulo 4 Trabalhos Anteriores 27
4.1 Introdução 27
4.2 Trabalhos anteriores 27
4.3 Conclusão 34
Capítulo 5 Algoritmos 35
5.1 Introdução 35
5.2 Entradas dos algoritmos 36
5.2.1 Especificação do Sistema 36
5.2.2 Taxa de recepção dos dados 38
5.2.3 Biblioteca de Componentes 38
5.3 Alocação, Mapeamento e Escalonamento em Estruturas Pipeline 40
5.3.1 Particionamento *hardware-software* 41
5.3.2 Cálculo das distâncias 43
5.3.3 Escalonamento 45
5.3.4 Alocação, mapeamento e escalonamento das tarefas 47
5.3.4.1 Síntese de Pipeline com Minimização dos Processadores do Sistema (MPS) 47
5.3.4.2 Síntese de Pipeline com Minimização do Custo do Sistema (MCS) 52
5.3.4.3 Síntese de Pipeline com Minimização da Latência do Sistema (MLS) 57
5.4 Conclusão 66
Capítulo 6 Resultados Experimentais 67
6.1 Ambiente de Desenvolvimento e Testes 67
6.2 Sistemas Especificados por Grafos Sintéticos 67

6.2.1 Testes usando grafos sintéticos com menos de 20 tarefas 67
6.2.2 Sistemas especificados por grafo com 50 tarefas 68
6.3 Compressor de sinal digital de áudio (AC3) 72
6.4 Comparação com trabalhos anteriores 76
6.5 Conclusões 80
Capítulo 7 Contribuições e trabalhos futuros 81
7.1 Contribuições 81
7.2 Trabalhos futuros 82
Referências Bibliográficas 85
Apêndice A 89
A.1 Testes usando grafos sintéticos com menos de 20 tarefas 89
A.2 Conclusão 92
Apêndice B 99
C.1 Definição do grafo de tarefas 99
C.2 Biblioteca de Componentes 101
Apêndice C 105
C.1 Algoritmo – Minimização do Custo do Sistema (MCS) 105
C.2 Algoritmo – Minimização do Número de Processadores do Sistema (MPS) 106
C.3 Algoritmo – Minimização da Latência do Sistema (MLS) 108
C.4 Algoritmo MLS - Gera agrupamento das tarefas (taxa de utilização) 110
C.5 Escalonamento das Tarefas 112

Lista de figuras

Figura 1.1 – Modelos de implementação em *pipeline* 4
Figura 2.1 - Arquitetura básicas de um processador [20] 8
Figura 2.2 - Arquitetura básica do processador genérico [20] 10
Figura 2.3 - Arquitetura básica do processador dedicado [20] 12
Figura 2.4 - Arquitetura básica do processador específico para uma área de aplicação [20] 13
Figura 3.1 - Metodologia genérica para síntese de sistemas embutidos [8] 19
Figura 3.2- Exemplo de uma MEF com 3 estados 22
Figura 3.3 - Exemplo de grafo de fluxo de dados (a) e grafo de fluxo de controle (b) 23
Figura 3.4 - Exemplo de diagrama de bloco de sistema 24
Figura 3.5 - Exemplo de um diagrama de entidade-relacionamento (DER) 24
Figura 3.6 - Exemplo de diagrama de fluxo de dados e controle (DFDC) 24
Figura 5.1 - Grafo de tarefas 37
Figura 5.2 - Etapas dos algoritmos de síntese de sistemas embutidos 42
Figura 5.3 – Caminho do grafo a partir da tarefa T1 44
Figura 5.4 - Distribuição das tarefas no pipeline (a) e a distribuição no tempo (b) 46
Figura 5.5 – Escalonamento das tarefas aplicando o algoritmo MPS 51
Figura 5.6 - Grafo de tarefas (a), Biblioteca de Componentes (b), cálculo das distâncias (c) e o escalonamento final do sistema (d) 54
Figura 5.7 – Escalonamento do grafo na Figura 5.1, usando a Biblioteca de Componentes, aplicando o algoritmo MCS 57
Figura 5.8 – Exemplo de síntese (c) de um grafo (a) com dependência entre tarefas de diferentes grupos. A Biblioteca de Componentes usada para a síntese é apresentada em (b). 60

Figura 5.9 – Exemplo de um grafo (a) sem agrupamento das tarefas e (b) com agrupamento da tarefas 63

Figura 5.10 - Escalonamento para o sistema da Figura 5.9(a) após fase 1 63

Figura 5.11 - Escalonamento para o sistema da da Figura 5.9(a) após fase 2 65

Figura 6.1 – Parâmetros de avaliação do projeto para os *pipelines* sintetizados pelos algoritmos: (a) custo, (b) número de processadores e (c) número de estágios. 71

Figura 6.2 – Tempo de execução dos algoritmos para a síntese dos *pipelines*. 73

Figura 6.3 - Grafo do compressor de áudio AC3 74

Figura 6.4 - Grafo de tarefas [2] 77

Figura 6.5 - Escalonamento obtido pelo algoritmo apresentado no artigo [2] (a) e pelos algoritmos MPS (b), MCS (c) e MLS (d). 79

Figura A.1 – Grafo de teste com 11 tarefas [15] 91

Figura A.2 – Parâmetros de avaliação do projeto para os *pipelines* sintetizados pelos algoritmos para o grafo da Figura A.1: (a) custo, (b) número de processadores, (c) número de estágios e (d) taxa de ocupação dos processadores. 94

Figura A.3 – Grafo de teste com 17 tarefas [15] 95

Figura A.4 – Parâmetros de avaliação do projeto para os *pipelines* sintetizados pelos algoritmos para o grafo da Figura A.3: (a) custo, (b) número de processadores, (c) número de estágios e (d) taxa de ocupação dos processadores. 97

Figura B.1 – Definição do grafo de tarefas 100

Lista de Tabelas

Tabela 5-1 – Informações da Biblioteca de Componentes 40
Tabela 5-2 - Cálculo da distância das tarefas.. 45
Tabela 6-1 - Intervalo de variação dos tempos de execução.. 69
Tabela 6-2- Cálculo dos tempos de comunicação... 75
Tabela 6-3 - Biblioteca de Componentes.. 75
Tabela 6-4 - Métricas de projeto para os *pipelines* do compressor de áudio AC3 76
Tabela 6-5 - Biblioteca de Componentes.. 77
Tabela 6-6 - Métricas de projeto... 80
Tabela A-1 – Biblioteca de Componentes (tempos de execução em ms).. 90
Tabela B-1 – Biblioteca de Componentes 102
Tabela B-2 – Biblioteca de Componentes (cont.)... 103

Glossário

ASIC - Application-Specific Integrated Circuit

ASIP - Application-Specific Instructor Set Computer

ATM - Asynchronous Transfer Mode

ATSC – United States Advanced Television Systems Committee

CISC – Complex Instruction-Set Computer

DSP – Digital Signal Processing

EST – Earliest Start Time

FPGA – Field Programmable Gate Array

IP – Internet Protocol

PC – Personal Computer

PCM – Pulse Code Modulation

RISC – Reduced Instruction-Set Computer

SOC – System on a Chip

STG – Standard Task Graph

Capítulo 1

Introdução

1.1 Motivação

O constante aumento na capacidade de processamento e a diminuição dos custos de produção dos processadores têm permitido empregá-los no desenvolvimento de uma grande variedade de sistemas digitais. Muitos destes sistemas são projetados para executarem funções específicas, que fazem parte de diferentes dispositivos, e recebem o nome de sistemas embutidos [5] [21] [22].

Os sistemas embutidos estão presentes atualmente em diferentes áreas da indústria executando funções como controle de processos, processamento digital de sinais, processamento de imagens, dentre outras. Para atender a diferentes requisitos de desempenho, tamanho, consumo de energia, custo e portabilidade, os sistemas embutidos muitas vezes adotam arquiteturas distribuídas formadas por processadores programáveis e outros elementos de *hardware*, tais como ASIC e FPGA [1] [14].

Para algumas aplicações, como por exemplo, codificadores e decodificadores de áudio e vídeo, cujo projeto inclui como especificação de desempenho, o intervalo mínimo entre os conjuntos de dados de entrada, adotar uma estrutura em *pipeline* de processadores pode possibilitar ao projetista obter uma solução com um menor custo.

1.2 Objetivo

O objetivo deste trabalho foi o desenvolvimento de algoritmos para a síntese de sistemas digitais heterogêneos, isto é, compostos por diferentes processadores, usando uma estrutura em *pipeline* de processadores para a execução das tarefas. Os algoritmos realizam a alocação de processadores, o mapeamento de tarefas nos processadores alocados e o escalonamento das tarefas, atendendo à restrição de desempenho dos sistemas representada pela taxa máxima de chegada dos conjuntos de dados de entrada e, ao mesmo tempo, procuram otimizar diferentes parâmetros de qualidade dos sistemas.

Considerando sistemas especificados por grafos acíclicos de tarefas e por uma taxa máxima de chegada dos conjuntos de dados de entrada, foram desenvolvidas três abordagens de síntese de *pipelines* para implementação dos sistemas, visando minimizar o número de processadores, o custo do sistema e a latência total dos sistemas.

Em todos os casos foram utilizados algoritmos iterativos baseados em lista de tarefas prontas. Para minimizar o número de processadores, o algoritmo considera a substituição de processadores alocados sempre que um novo processador precisa ser adicionado ao sistema. Para reduzir o custo, a alocação de novos processadores é feita com base numa estimativa de custos do sistema. Por fim, o algoritmo que minimiza o número de estágios e conseqüentemente a latência, procura reduzir tanto o tempo de execução das tarefas como os tempos de comunicação entre elas.

A próxima seção descreve o funcionamento de um *pipeline* de processadores e as métricas utilizadas para avaliar seu desempenho.

1.3 Pipeline

Como mencionado na seção anterior, o modelo em *pipeline* é apropriado para a implementação de sistemas que processam seqüências de conjuntos de dados, pois permite a execução das funções do sistema de forma concorrente, distribuindo-as em estágios.

O tempo máximo de processamento num estágio do *pipeline*, denominado *latência do estágio*, deve ser menor ou igual ao inverso da taxa máxima de chegada dos dados, permitindo que os dados sejam processados à medida que estejam disponíveis [9]. Durante a execução do sistema, os dados são passados de estágio em estágio até que sejam completamente processados e os resultados finais tenham sido gerados. O *pipeline* permite alcançar taxas de processamento mais altas do que se conseguiria com uma implementação que execute todas as funções num único estágio, pois cada estágio pode tratar um conjunto diferente de dados.

A Figura 1.1, apresenta o funcionamento de um *pipeline* de 4 estágios e latência τ. Durante o intervalo $0 - \tau$, o primeiro conjunto de dados, D1, é processado pelo primeiro estágio do *pipeline*. Durante o intervalo $\tau - 2\tau$, o primeiro conjunto de dados é passado para o estágio 2, enquanto o próximo conjunto de dados, D2, começa a ser processado pelo estágio 1. No próximo intervalo, $2\tau - 3\tau$, D1 passa para o estágio 3, D2 passa para o estágio 2 e um novo conjunto de dados começa a ser processado. Finalmente, no intervalo $3\tau - 4\tau$, os dados de D1 serão processados pelo último estágio, enquanto os outros dados continuam avançando nos estágios do *pipeline* e um novo conjunto de dados D4 inicia seu processamento no primeiro estágio. Os resultados referentes ao primeiro conjunto de dados de entrada, D1, são obtidos em 4τ e os resultados referentes aos conjuntos subseqüentes de dados de entrada serão gerados ao final de cada intervalo de duração τ. Desta forma, a estrutura *pipeline* possibilita um aumento da produtividade, isto é, do número de conjuntos de dados processados por unidade de tempo e de resultados em relação a uma implementação *não-pipeline*.

1.3.1 Métricas de desempenho

As métricas de desempenho de um sistema podem ser relacionadas ao tempo necessário para o sistema ser executado. As métricas podem ser classificadas de acordo com a unidade de medida usada no cálculo da métrica [9]. Geralmente, três tipos de unidades são usados: ciclos de relógio, passos de controle e tempo de execução.

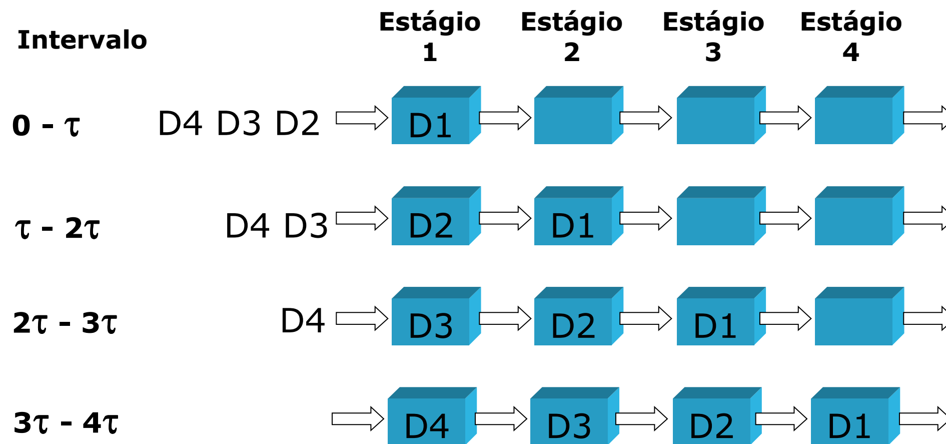


Figura 1.1 – Modelos de implementação em *pipeline*

Os ciclos de relógio afetam o tempo de execução da aplicação e os recursos usados para implementar o sistema, uma vez que determinadas tecnologias especificam uma freqüência máxima para operação. Um passo de controle corresponde a um estado da unidade de controle do sistema, no qual as operações são executadas. O número de passos da unidade de controle afetam a complexidade da lógica de controle do sistema [9]. Estas métricas dependem diretamente da tecnologia do processador, conforme será visto na seção 2.

O tempo de execução é o tempo médio para execução das funções de um sistema do início até seu término. Duas outras importantes métricas de desempenho associadas ao tempo de execução de sistemas com estruturas em *pipeline* são a produtividade*,* ou *throughput do pipeline* e a *latência total* do sistema [9].

A produtividade do *pipeline* mede a freqüência com que os dados são gerados pelo sistema sendo igual a:

$$produtividade = \frac{1}{latência_do_estágio} \qquad \text{(eq. 1.1) [9]}$$

Os resultados referentes ao processamento do primeiro conjunto de dados são gerados somente após seu processamento pelo último estágio do *pipeline*. Assim, a produtividade corresponde ao número de conjuntos de dados processados por unidade de tempo, após o término do processamento do primeiro conjunto.

A transmissão de resultados entre estágios na estrutura em *pipeline* ocorre no fim do processamento associado ao estágio e, portanto, em intervalos iguais à latência do estágio. Assim, o tempo de execução do *pipeline* é sempre calculado considerando-se a latência dos estágios e não o tempo de execução das funções do sistema.

A última métrica, tempo total de processamento de um conjunto de dados no sistema, denominado *latência total do sistema*, mede o tempo necessário para que os dados passem por todos os estágios do *pipeline*. Em um *pipeline* cujo número de estágios é referenciado como *número_de_estágios*, a *latência total do sistema* (*latência_total*) é igual a:

$$latência_total = número_de_estágios \times latência_do_estágio \qquad \text{(eq 1.2) [9]}$$

A *latência do estágio* do *pipeline*, referenciada como *latência_do_estágio*, é uma métrica importante para o projeto do sistema, sendo definida pela taxa de chegada dos dados especificados. Outro fator determinante da latência do sistema é o número de estágios que é influenciado pelo tempo de execução e pela dependência entre as funções do sistema, uma vez que qualquer função deve iniciar e terminar sua execução dentro do mesmo estágio.

1.4 Organização da tese

Este trabalho está organizado em 7 capítulos.

Neste capítulo foi abordada a síntese de sistemas embutidos que realizam o processamento de conjunto de dados de entrada que chegam separados por pequenos intervalos de tempo e o uso da organização em *pipeline* nestes sistemas.

O Capítulo 2 apresenta uma descrição das principais tecnologias de processadores empregadas nos projetos de sistema embutidos. O Capítulo 3 apresenta uma visão geral dos sistemas embutidos, as principais etapas envolvidas em seus projetos e uma descrição de algumas metodologias desenvolvidas para auxiliar no projeto destes sistemas. No Capítulo 4, apresentamos uma análise de alguns dos trabalhos já publicados sobre metodologias para a síntese de sistemas embutidos. No Capítulo 5, descrevemos detalhadamente os algoritmos desenvolvidos com exemplos da aplicação de cada um deles. O Capítulo 6 apresenta os resultados experimentais dos algoritmos para sistemas especificados por grafos sintéticos e também para uma aplicação real. O Capítulo 7 apresenta as conclusões, bem como as contribuições deste trabalho e sugestões para futuros trabalhos.

Capítulo 2

Tecnologia de Processadores

O objetivo deste capítulo é apresentar uma descrição geral das principais tecnologias adotadas nos projetos de processadores empregados na síntese de sistemas embutidos. No contexto deste trabalho, tecnologia de um processador refere-se à sua arquitetura interna que, por sua vez, está relacionada à organização de seus módulos, à sua capacidade de executar uma determinada função do sistema e à estrutura de interligação entre os módulos [20].

2.1 Organização básica de um processador

A tecnologia de um processador pode variar conforme seu grau de especialização e sua finalidade. Uma característica dos processadores que varia com sua especialização é sua programabilidade, que define, entre outros aspectos, como os programas são armazenados e sua capacidade para executá-los. Segundo estes critérios, os processadores podem ser classificados como processadores genéricos, processadores dedicados e processadores específicos para uma área de aplicação.

Todas estas classes de processadores compartilham de uma organização básica que é constituída de duas partes principais: a unidade de controle (UC) e o *datapath* (DP) [12], conforme podemos ver na Figura 2.1. Estas partes são formadas por diferentes componentes interligados por barramentos, responsáveis pela transmissão de instruções, dados e sinais de controle.

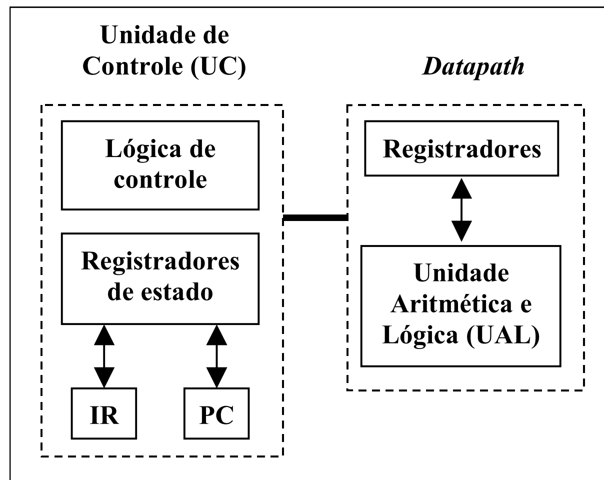


Figura 2.1 - Arquitetura básicas de um processador [20]

A unidade de controle tem a função de controlar e coordenar a execução dos programas no processador, buscando as instruções na memória, determinando seu tipo e enviando os sinais de controle para o *datapath* para sua execução. A unidade de controle é constituída de um conjunto de circuitos, denominado lógica de controle, um conjunto de registradores de estado usados para controlar o estado interno do processador durante a execução das instruções e dois outros registradores específicos usados na busca e decodificação das instruções. O registrador denominado contador de programa (PC) armazena o endereço na memória da próxima instrução a ser executada e o registrador de instrução (IR) armazena o código da instrução.

Para ser executada pelo processador, uma instrução precisa passar por uma série de passos intermediários, conhecidos como ciclo de leitura-decodificação-execução. De forma resumida, tais passos são os seguintes [19]:

1. Leitura da próxima instrução na memória e armazenamento da instrução no IR;
2. Atualização do valor do PC, fazendo-o apontar para a próxima instrução a ser executada;
3. Determinação do tipo de instrução armazenada em IR;
4. Se a instrução precisar de um dado armazenado na memória, neste passo deve ser calculado o endereço deste dado;
5. Leitura do dado e armazenamento em um dos registradores do processador;

6. Execução da instrução;
7. Retorno ao passo 1 para iniciar o ciclo correspondente à próxima instrução armazenada na posição especificada pelo contador de programa (PC).

O *datapath* é a parte do processador com a função de executar as instruções dos programas, sendo constituído de unidades lógicas e aritméticas (ULA) onde estão os circuitos e unidades funcionais (somadores, multiplicadores, operadores lógicos, entre outros) que efetivamente executam as instruções e um conjunto de registradores onde os dados usados pelas operações e os resultados são armazenados temporariamente antes de serem transferidos para a memória por outras instruções. As unidades funcionais, assim como a quantidade e o tamanho dos registradores variam entre os processadores e são definidos no projeto do processador. O conjunto de instruções do processador define os projetos da unidade de controle e do *datapath.*

A seguir são descritas as características das três classes de processadores, destacando as diferenças com relação à arquitetura básica descrita acima.

2.2 Processadores genéricos

Os processadores genéricos são projetados para executar diversas aplicações e apresentam um alto grau de programabilidade. Uma característica importante deste processador é possuir um *datapath* genérico o suficiente para executar instruções úteis a uma larga gama de aplicações. Tipicamente, estes processadores têm uma quantidade grande de registradores e uma ou mais ULAs de propósito geral, com unidades funcionais capazes de executar diversas operações aritméticas (por exemplo, soma, subtração, divisão, multiplicação), lógicas (por exemplo, operações lógicas E, OU, Negação e comparações lógicas) e de deslocamento de bits, de modo que possam ser utilizados por programas desenvolvidos para diferentes finalidades [20].

Outra característica do processador genérico é que os programas e os respectivos dados são armazenados em memória, como mostra a Figura 2.2. O armazenamento em memória de programas e dados possibilita a execução de mais de um programa dentro de uma aplicação ou sistema. O modo como os programas e os dados são armazenados na memória define a arquitetura de memória que

pode ser de dois tipos. Na arquitetura de memória Princeton, os espaços de endereçamento são compartilhados e na arquitetura Harvard, os programas são armazenados em memória separada da memória usada pelos dados [20].[1]

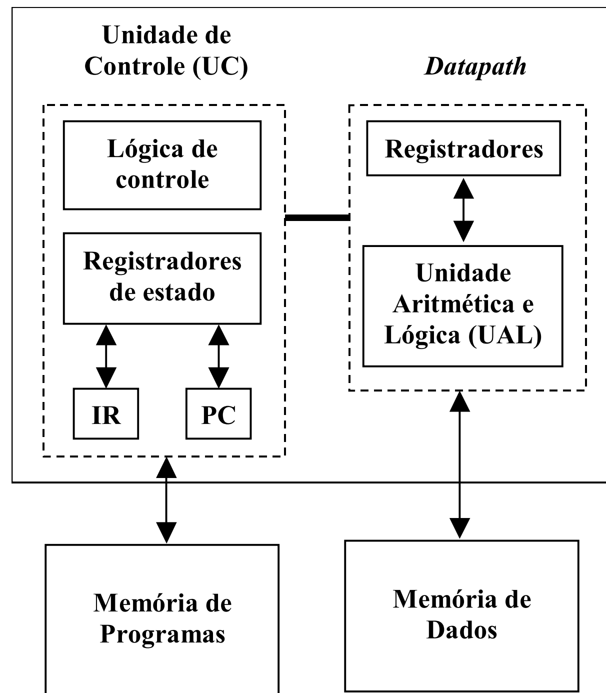


Figura 2.2 - Arquitetura básica do processador genérico [20]

Uma característica dos processadores de propósito geral é que as instruções, em sua maioria, não são otimizadas para uma classe de aplicações em particular. Desta forma, o tempo de execução destas aplicações pode ser menor em implementações que utilizam processadores dedicados ou específicos para uma área de aplicação, que serão apresentados nas próximas sub-seções.

[1] Nesta seção, as memórias de programas e de dados são mostradas separadamente para facilitar a apresentação dos conceitos. Caso a diferenciação entre as arquiteturas de memória seja necessária, ela será feita explicitamente.

2.3 Processadores dedicados

Os processadores dedicados são circuitos digitais específicos projetados para executar uma aplicação. Estes processadores não apresentam uma memória para armazenar os programas, pois são projetados para executar uma aplicação única cuja lógica é implementada diretamente na unidade de controle. Como não há necessidade de ler as instruções na memória, o registrador de instruções (IR) e o contador de programa (PC) também não fazem parte desta implementação. O *datapath*, ao contrário do encontrado nos processadores genéricos, contém somente os componentes necessários para a aplicação, como por exemplo, registrados e somadores, conforme pode ser visto na Figura 2.3. A memória de dados é usada para armazenar os dados necessários e os resultados dos cálculos [20].

Um programa que esteja armazenado na memória de programa e que seja executado pelo processador é referenciado como uma implementação em *software*, em oposição aos programas e aplicações que executam nos processadores dedicados que são referenciados como uma implementação em *hardware*.

A especialização destes processadores permite a otimização do projeto da unidade de controle e do *datapath* resultando numa implementação do processador específica para a aplicação com um desempenho superior à implementação obtida com os processadores genéricos, além do tamanho reduzido e menor consumo de energia. Entretanto, o custo deste tipo de processador, muitas vezes, é superior aos processadores genéricos, como conseqüência da tecnologia necessária para sua fabricação e da sua limitada flexibilidade, uma vez que, em caso de alteração na lógica da aplicação, é necessário alterar o projeto do processador [20].

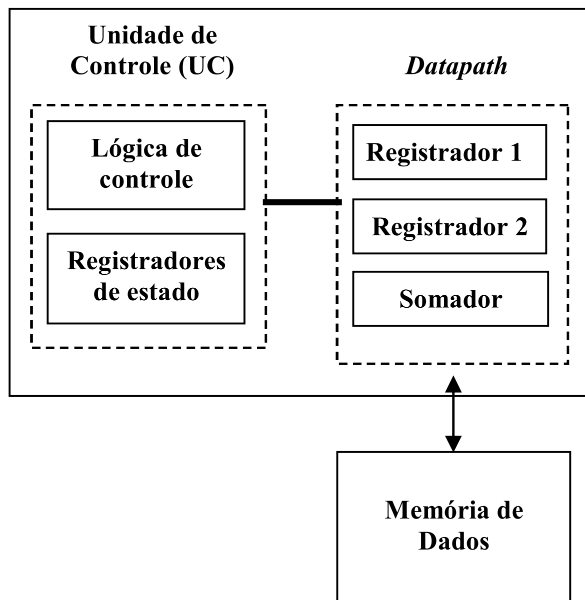


Figura 2.3 - Arquitetura básica do processador dedicado [20]

2.4 Processadores específicos para uma área de aplicação

Um processador específico para uma área de aplicação, também conhecido como *application-specific instructor set computer* (ASIP), é um processador programável direcionado para uma classe de aplicações, por exemplo, controle embutido, processamento digital de sinais e telecomunicações. O *datapath* é especializado para a classe de aplicações para a qual o processador é projetado. A especialização é realizada incluindo no projeto do processador unidades funcionais dedicadas que executam operações freqüentes na classe de aplicação visada e excluindo outras unidades funcionais menos utilizadas, conforme mostra a Figura 2.4. Os ASIPs se apresentam como um compromisso entre os dois tipos de processadores anteriormente apresentados e seu uso nos sistemas embutidos pode trazer a mesma flexibilidade encontrada nos processadores genéricos, mantendo as vantagens de bom desempenho, tamanho reduzido, baixo consumo de energia e dissipação de calor dos processadores dedicados.

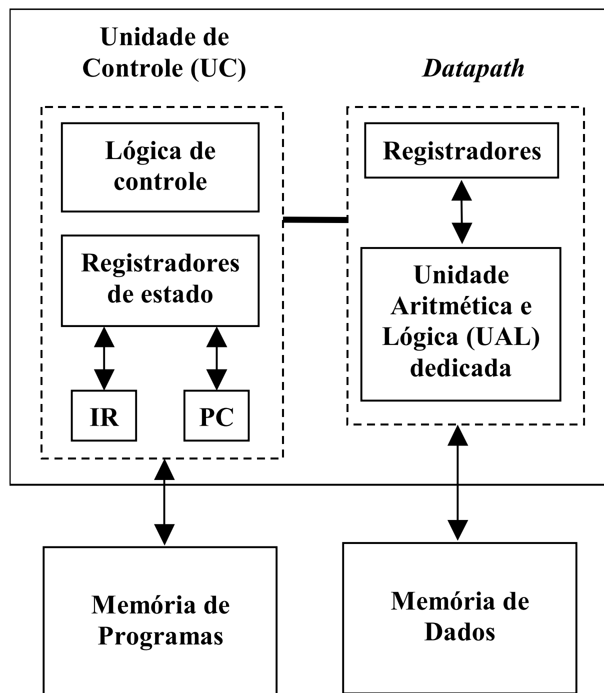


Figura 2.4 - Arquitetura básica do processador específico para uma área de aplicação [20]

Microcontroladores e processadores digitais de sinais (DSP) são exemplos conhecidos de ASIP que têm sido largamente utilizados há algumas décadas nos projetos de sistemas embutidos. O microcontrolador é um microprocessador destinado a aplicações embutidas de controle que tipicamente monitoram e alteram os valores de vários sinais de controle. Estas aplicações não demandam grande poder computacional, uma vez que não realizam uma quantidade grande de cálculos. Assim, seus *datapaths* tendem a realizar instruções simples para manipulação de bits e para escrita e leitura de bits em dispositivos externos aos processadores. Além disto, os microcontroladores incorporam diferentes componentes comuns a aplicações de controle, como controladores para comunicação serial de dados, temporizadores, contadores e conversores analógicos-digitais [20].

O DSP é um microprocessador projetado para executar operações comuns em processamento digital de sinais como a codificação digital de sinais analógicos de áudio e vídeo. As instruções, que envolvem filtragem, transformação e combinação de sinais, são matematicamente intensas e incluem

operações combinadas como soma e multiplicação ou soma e deslocamento. O *datapath* possui componentes específicos para a execução destas instruções. Outra característica das aplicações DSP é a manipulação de grandes vetores de dados. Para reduzir o tempo de execução, são incluídos componentes de hardware para realizar a escrita e leitura seqüencial de dados na memória em paralelo com outras operações.

2.5 Conjunto de instruções

Uma outra maneira de classificar os processadores programáveis é com relação ao seu conjunto de instruções. Segundo este critério, um processador pode ser classificado como **processador com um conjunto complexo de instruções**, (CISC), ou como **processador com um conjunto reduzido de instruções**, (RISC).

2.5.1 Processadores CISC

Podemos citar duas motivações para o desenvolvimento dos processadores CISC. A primeira motivação consiste de reduzir o número de instruções no código compilado, minimizando o número de acessos à memória para buscar as instruções. A segunda motivação foi simplificar a construção de compiladores, incluindo instruções complexas no conjunto de instruções que imitassem as sentenças de uma linguagem de programação de alto nível.

As instruções possuem tamanhos diferentes com tempos diferentes de execução. Para implementar um conjunto complexo de instruções, o *datapath* do processador tem de ser complexo e a unidade de controle geralmente é microprogramada, ou seja, cada instrução do conjunto de instruções é sub-dividida em micro instruções que são interpretadas pelo processador para execução da instrução. Os dados necessários à execução das instruções podem ser originados de diferentes fontes, memória ou registradores, e com tamanhos variáveis, obrigando a implementação de diferentes tipos de dados e de modos de endereçamento para acessar os dados, que podem ser, por exemplo, imediato, direto, indireto e indexado.

2.5.2 Processadores RISC

Em contraste com os processadores CISC, a arquitetura RISC é otimizada para diminuir o tempo de execução das instruções e reduzir a extensão do ciclo de relógio. De maneira geral, o *datapath* de um processador RISC possui uma grande quantidade de registradores e ALUs. Os registradores são usados para armazenar todos os dados necessários para as instruções e seus resultados, minimizando o número de acesso à memória para busca dos dados. Os modos de endereçamento e as instruções para leitura e escrita dos dados em memória são simplificadas. Em muitos casos, resume-se a instruções de *load* e *store*. Este modo de acesso à memória simplifica o projeto do *datapath*.

A simplicidade da arquitetura e das instruções, entretanto, requer um compilador mais sofisticado. Além disto, como o número de instruções é reduzido, às vezes, é preciso reunir uma seqüência de instruções RISC para implementar uma operação equivalente a uma única instrução CISC. Em média, programas RISC tendem a ocupar de 20% a 30% mais memória que programas CISC [8].

2.6 Conclusão

Neste capítulo apresentamos as descrições das tecnologias dos processadores utilizados nos projetos de sistemas embutidos. Os processadores compartilham uma arquitetura básica e apresentam diferentes níveis de especialização que os resultam em diferentes classes, incluindo processadores genéricos, processadores específicos para uma área de aplicação e processadores dedicados. A especialização implica em diferenças nas implementações da unidade de controle e do *datapath* dos processadores, possibilitando a adequação dos mesmos a uma finalidade específica. Estas diferenças podem ser na quantidade ou na organização dos componentes que formam estes elementos.

Um sistema embutido heterogêneo pode ser composto de processadores com diferentes graus de especialização, empregados para atender aos diferentes requisitos como custo, desempenho e tamanho. A seleção dos processadores que compõem um sistema é uma das etapas do processo de

síntese e deve considerar diferentes fatores. Em muitas situações, os processadores genéricos que podem ter um custo mais baixo e podem ser utilizados para implementação da maioria das funções do sistema, enquanto que os processadores mais específicos, que apresentam um custo maior, são empregados nas rotinas cujos requisitos de desempenho não podem ser atendidos pelos processadores genéricos.

No próximo capítulo, abordaremos as tecnologias adotadas nos projetos de processadores e descreveremos as principais características do projeto de sistemas embutidos.

Capítulo 3

Projeto de Sistemas Embutidos

Este capítulo apresenta conceitos básicos sobre sistemas embutidos e módulos usados na sua especificação e técnicas de projeto.

3.1 Sistemas Embutidos

Os sistemas embutidos estão presentes atualmente em diferentes áreas da indústria executando funções em controle de processos, processamento digital de sinais, processamento de imagens, dentre outras. Exemplos de sistemas embutidos incluem [9] [18] [21]:

- Aplicações simples, como fornos de microondas, impressoras e modems que requerem componentes de baixo custo;
- Dispositivos portáteis, como aparelhos celulares em que os sistemas embutidos, que realizam funções de processamento de sinais e outras tarefas sofisticadas têm o consumo de energia e o tamanho como restrições críticas de projeto;
- Consoles de *video game* que exigem excelente desempenho na manipulação de gráficos;
- Controladores industriais, para os quais requisitos de confiabilidade, manutenabilidade e facilidade de programação são fundamentais;
- Aplicações em telecomunicações, como comutadores ATM, roteadores para rede IP, estações base para comunicação móvel;
- Sistemas de controle automotivos;

- Aplicações de codificação e decodificação de áudio e vídeo.

Os sistemas embutidos podem variar em complexidade, conforme pode ser observado nos exemplos acima. Independentemente de sua complexidade, os sistemas embutidos são específicos para executar uma determinada função dentro do sistema onde estão inseridos. Estas aplicações são funções reativas que demandam respostas em tempo-real a eventos ocorridos no ambiente.

Devido à natureza das aplicações, os sistemas embutidos apresentam diferentes restrições em termos de, por exemplo, confiabilidade, desempenho, tamanho, consumo de energia, portabilidade e custo. Todos estes parâmetros devem ser considerados durante o projeto. É comum encontrar sistemas que adotam arquiteturas distribuídas consistindo de um ou mais processadores genéricos ou de propósitos específicos (ASIP) com diferentes características para atender a estas restrições [3] [15].

3.2 Projeto de Sistemas Embutidos

O projeto de sistemas embutidos freqüentemente requer o emprego de técnicas e metodologias de especificação e desenvolvimento diferentes das comumente usadas nos projetos de aplicações executadas em computadores de propósito geral. Uma importante característica do projeto de sistemas embutidos é a implementação das funções do sistema em componentes de *software* e de *hardware* , tornando mais fácil o atendimento aos requisitos do projeto [8] [21].

Esta característica tem motivado o desenvolvimento de metodologias e ferramentas de auxílio ao projeto, que abrangem linguagens e modelos de especificação de sistemas, ambientes de desenvolvimento e algoritmos de síntese de sistemas. Uma metodologia genérica para o desenvolvimento de sistemas embutidos envolve várias etapas [8], conforme mostra a Figura 3.1. As etapas do processo de síntese, indicadas pelos números entre parênteses, são comentadas a seguir.

A definição dos requisitos do sistema (1) é realizada diretamente pelo cliente ou pela equipe de marketing da empresa a partir das necessidades e restrições identificadas. Nesta etapa, as definições dos requisitos são feitas sem considerar os detalhes de implementação do sistema final.

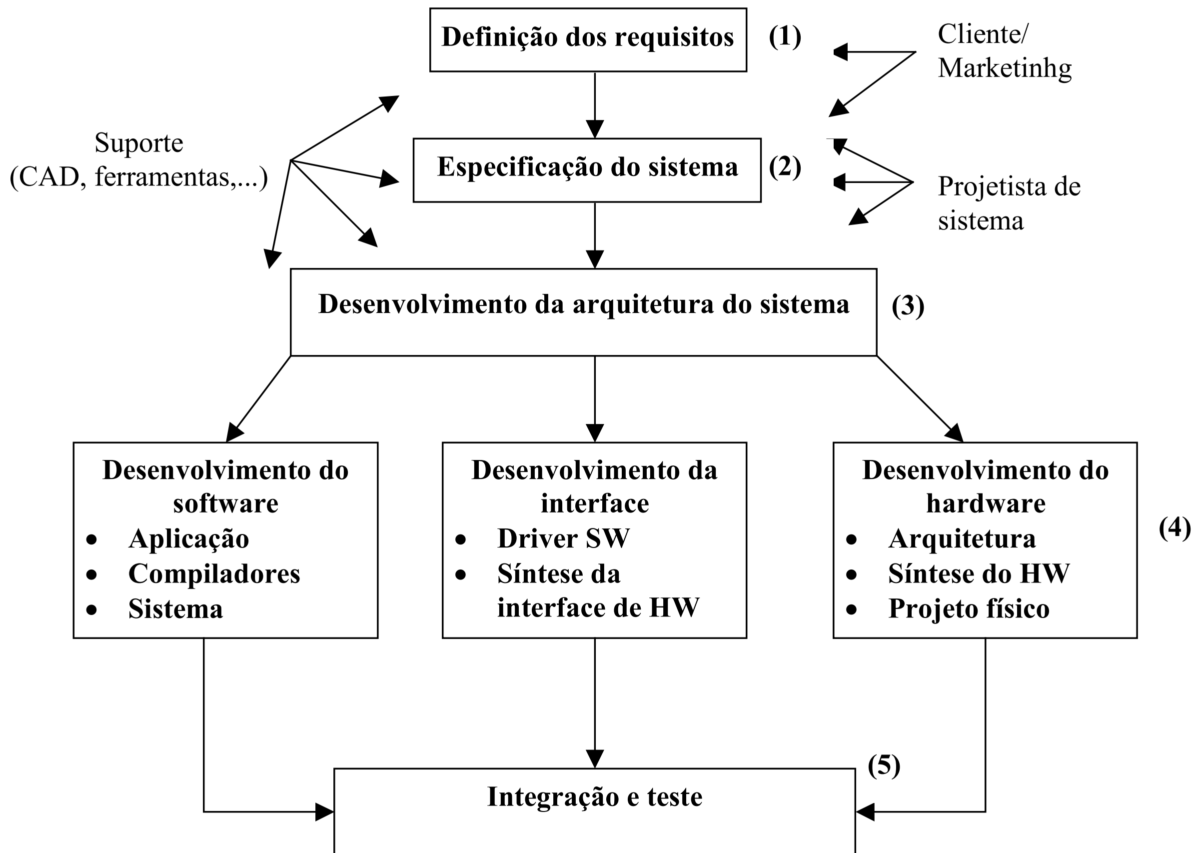


Figura 3.1 - Metodologia genérica para síntese de sistemas embutidos [8]

A especificação do sistema (2) é feita pelos projetistas do sistema a partir dos requisitos levantados na etapa anterior, adotando um determinado modelo para especificação destes requisitos, podendo ainda ser empregadas ferramentas de suporte para a geração da especificação do sistema que pode ser feita em diferentes níveis de abstração. No nível mais alto de abstração, temos as especificações que descrevem o sistema através das funções que este desempenha e, no nível mais baixo, temos as descrições do sistema como um conjunto de componentes físicos interligados. Algumas técnicas e modelos comumente usados para especificação dos sistemas serão discutidos na seção seguinte.

Os algoritmos de síntese de sistemas são empregados na etapa de desenvolvimento da arquitetura do sistema (3) e consideram a síntese dos módulos de *software* e *hardware* de forma integrada. A

síntese dos sistemas é dividida em sub-etapas que podem variar conforme a metodologia de síntese adotada. As principais sub-etapas da síntese são (a) particionamento das tarefas do sistema entre os módulos de *hardware* e *software*, (b) alocação dos elementos de processamento do sistema (processadores e elementos de *hardware* dedicados), (c) mapeamento das tarefas nos elementos de processamento alocados e (e) escalonamento das tarefas. A seqüência destas sub-etapas durante a síntese depende da metodologia ou ferramenta adotada para o projeto. Por exemplo, na síntese de sistemas onde os processadores do sistema são pré-definidos, a sub-etapa de alocação de processadores não é necessária.

Na etapa (4), ocorre o desenvolvimento dos módulos de *software* e *hardware* do sistema, bem como das interfaces entre eles. No desenvolvimento do *software*, é realizada a definição do sistema operacional e do compilador a serem usados para o desenvolvimento das aplicações. O desenvolvimento do *hardware* abrange o projeto dos componentes de *hardware* do sistema. O desenvolvimento das interfaces cobre o desenvolvimento dos *drivers* (*software*) e dos componentes de *hardware* que compõem as interfaces entre os módulos de *software* e *hardware*.

Na última etapa (5) é realizada a integração dos módulos de *hardware* e *software* desenvolvidos na etapa anterior e são realizados os testes funcionais destes módulos e das interfaces, nos quais é verificado se a implementação dos módulos está correta conforme a especificação. Por último, são realizados os teste de sistema, nos quais o sistema final é testado de forma integrada.

3.3 Especificação do sistema

A especificação do sistema é, muitas vezes, composta pela descrição das funções a serem desempenhadas e pelas restrições, por exemplo, de performance, tamanho e potência. O projeto de sistema é um processo de implementação destas funções usando um conjunto de componentes físicos, como processadores, memória e elementos dedicados de hardware.

O modelo do sistema, que consiste de objetos interligados de acordo com regras de composição próprias do modelo, tem o objetivo de fornecer uma visão abstrata adequada para a síntese. O modelo deve ser formal, de modo a não apresentar ambigüidades e ser completo para poder

descrever todas as características do sistema. Diferentes modelos ou representações do sistema podem ser usados pelo projetista nas diferentes etapas do projeto. Os modelos podem ser divididos em 5 categorias: orientados a estados; orientados a atividades; orientados a estrutura; orientados a dados e heterogêneos [9].

Um modelo orientado a estados representa os sistemas como um conjunto de estados e um conjunto de transições entre os estados que são ativados por eventos externos. Este modelo é adequado para a representação de sistemas de controle, onde o comportamento temporal do sistema é o aspecto mais importante da especificação do sistema. Um exemplo deste tipo de modelo é a máquina de estados finita (MEF) que consiste basicamente de um conjunto de estados, um conjunto de transições entre eles, um conjunto de ações associadas aos estados e às transições entre eles e um conjunto de estradas e saídas. A Figura 3.2 mostra uma MEF com 3 estados e com transições entre todos os estados.

Os modelos orientados a atividades, como o grafo de fluxo de dados (GFD) e o grafo de fluxo de controle (GFC), descrevem um sistema como um conjunto de atividades relacionadas pelos fluxos de dados que representam as dependências entre as tarefas nas suas execuções, sendo apropriados para descrição de sistemas transformacionais, como, por exemplo, processamento digital de sinais, onde os dados passam por um conjunto de transformações a uma taxa constante. Estes modelos têm sido muito empregados na especificação de sistemas embutidos ([1] [3] [5] [14] [15] [17]) por permitirem a representação das funções e dependências em um nível alto de abstração.

O GFD consiste de um conjunto de nós e de arestas, conforme apresentado na Figura 3.3-a. Os nós podem ser classificados em nós de entrada (nós inicias), de saída (nós de destino), que representam, respectivamente, as entradas e saídas dos dados, nós de atividades, que representam as funções do sistema, e nós de armazenamento, que representam as estruturas de armazenamento de dados do sistema.

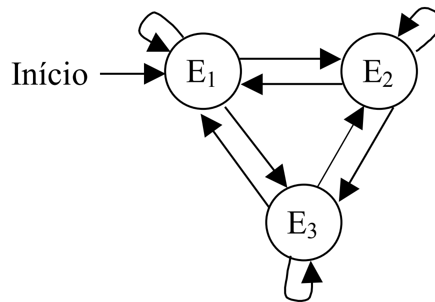


Figura 3.2- Exemplo de uma MEF com 3 estados

No GFC da Figura 3.3-b, os nós têm basicamente as mesmas atribuições dos nós do GFD, com o acréscimo do nó de decisão usado para controlar os desvios no fluxo de execução do grafo e os arcos representam o fluxo de controle do sistema.

Diferentemente dos modelos acima descritos que refletem as funcionalidades dos sistemas que estão representando, os modelos orientados a estrutura têm foco na descrição dos módulos físicos do sistema, apresentando, assim, uma visão estrutural do sistema. Os modelos orientados a estrutura podem representar os componentes físicos em diferentes níveis de abstração, como, por exemplo, o diagrama de bloco do sistema, mostrado na Figura 3.4, que apresenta os componentes que formam a arquitetura física de um sistema como os processadores e os módulos de memória. Outros tipos de representação são os diagramas em nível de registradores e em nível de portas lógicas, usados para representações em mais baixo nível dos componentes do sistema.

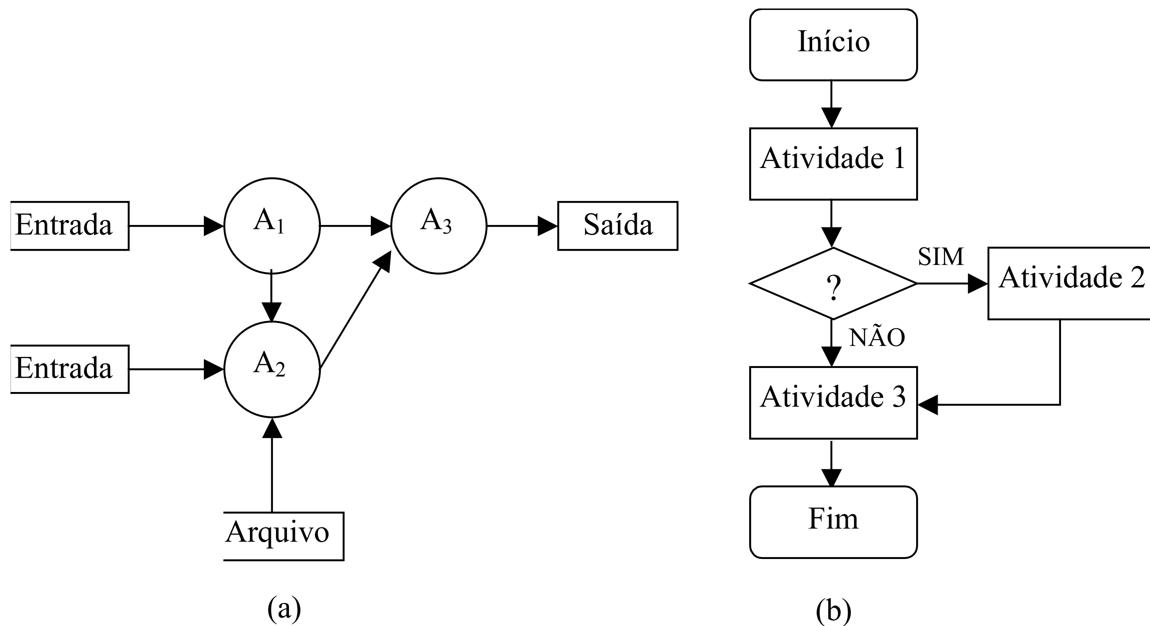


Figura 3.3 - Exemplo de grafo de fluxo de dados (a) e grafo de fluxo de controle (b)

Os modelos orientados a dados são usados para representar os sistemas como uma coleção de dados relacionados por seus atributos. Este modelo é apropriado para a representação de sistemas de bancos de dados, onde a representação da organização dos dados do sistema é o principal aspecto a ser descrito. Um exemplo desta classe de modelo é o diagrama de entidade-relacionamento (DER) que define o sistema como um conjunto de entidades, que representam os elementos de dados do sistema, os atributos destas entidades e os relacionamentos entre as entidades. A Figura 3.5 mostra um exemplo de um diagrama de entidade-relacionamento (DER) com duas entidades interligadas por um relacionamento. A entidade 1 apresenta dois atributos.

Por último, temos os modelos heterogêneos que podem agregar diferentes características dos outros modelos acima descritos para a representação dos sistemas. Como exemplos de modelos heterogêneos, podemos citar o grafo de fluxo de dados e controle (GFDC), mostrado na Figura 3.6 , e o *structure chart*.

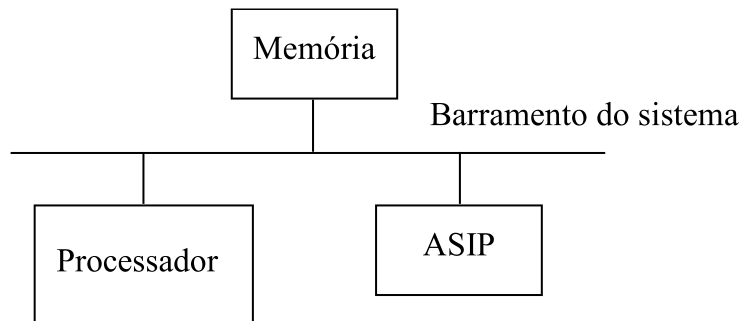


Figura 3.4 - Exemplo de diagrama de bloco de sistema

O GFDC combina as características do grafo de fluxo de dados, documentando o fluxo de dado entre as atividades e do grafo de fluxo de controle para controlar a seqüência de execução dos grafos numa única representação, sendo também muito usado na representação dos sistemas embutidos [7] [11] [16].

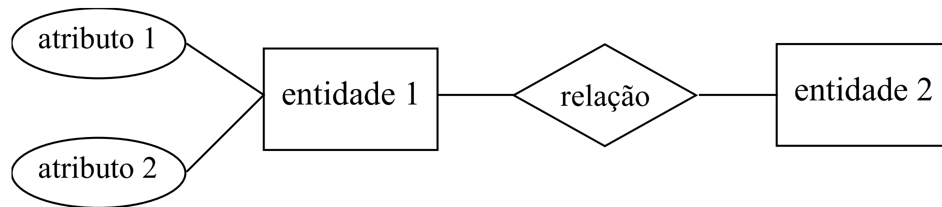


Figura 3.5 - Exemplo de um diagrama de entidade-relacionamento (DER)

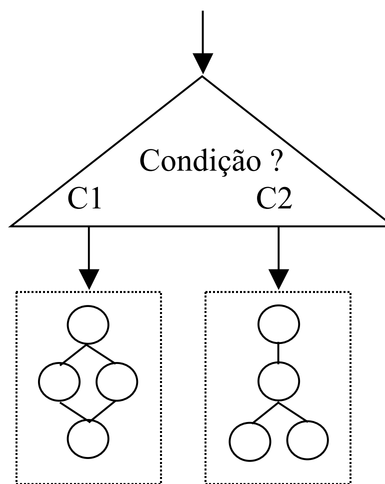


Figura 3.6 - Exemplo de diagrama de fluxo de dados e controle (DFDC)

3.4 Conclusão

Neste capítulo apresentamos uma visão geral das etapas de uma metodologia para o desenvolvimento de projetos de sistemas embutidos. As etapas abrangem a definição dos requisitos, a especificação do sistema através de um modelo, o desenvolvimento dos módulos e a integração final do sistema. Foram descritos, também, com maior detalhe os modelos que podem ser adotados para a especificação destes sistemas.

O capítulo seguinte descreve alguns dos trabalhos desenvolvidos na área de síntese de sistemas. Os algoritmos apresentados realizam a síntese, a partir de uma especificação que geralmente inclui as funções a serem desempenhadas pelo sistema e várias características não-funcionais a serem satisfeitas.

Capítulo 4

Trabalhos Anteriores

Este capítulo tem por objetivo descrever alguns dos algoritmos encontrados na literatura para a síntese de sistemas embutidos. Alguns destes algoritmos, da mesma forma que o trabalho desenvolvido nesta dissertação, geram sistemas estruturados como um *pipeline* de processadores.

4.1 Introdução

Ao longo das últimas décadas, muitos trabalhos vêem sendo desenvolvidos na área de síntese de sistemas embutidos. Estes trabalhos descrevem metodologias que abrangem as etapas do processo de síntese apresentadas no capítulo 3 e aplicam diferentes abordagens para a síntese de sistemas embutidos, homogêneos e heterogêneos, minimizando parâmetros de projetos. Os sistemas sintetizados são classificados como homogêneos, quando compostos por processadores de um único tipo, ou heterogêneos, quando os processadores podem ser de diferentes tipos.

4.2 Trabalhos anteriores

Nos trabalhos em [1] e [2], Bakshi e Gajski apresentam um algoritmo iterativo para a síntese da especificação funcional de um sistema, representada por um grafo de fluxo de dados, numa estrutura de *pipeline* de processadores. Os algoritmos apresentados nestes trabalhos têm como objetivo atender à restrição de desempenho da aplicação, representada pela taxa de chegada dos dados

enquanto minimizam os custos dos sistemas. A latência do estágio do *pipeline,* determinada no início do processo de síntese, é determinada pela taxa máxima de chegada dos dados.

O particionamento das tarefas em *hardware* (ASIC) e *software* (processadores), a alocação dos processadores e o escalonamento em *pipeline* são feitos em etapas sucessivas do algoritmo. As tarefas são implementadas preferencialmente por módulos de *software*, requerendo o desenvolvimento de componentes de *hardware* dedicado somente às tarefas cujas implementações em *software* apresentam tempos de execução maior que a latência do estágio em todos os processadores disponíveis. Para o particionamento das tarefas, o algoritmo utiliza uma base de dados que contém informações sobre as implementações para as tarefas do sistema em diferentes processadores e em *hardware*.

Após o particionamento das tarefas, é realizada a alocação inicial dos processadores, considerando a alocação mais barata que consiste de uma instância do processador mais barato que aloca todas as tarefas respeitando a restrição da latência do estágio.

A próxima etapa é o escalonamento das tarefas. Se o escalonamento não for possível na alocação definida, uma nova alocação de processadores é realizada, seguida de um novo escalonamento. Este processo é repetido até que seja encontrada uma alocação para o sistema que permita o escalonamento das tarefas.

Um dos algoritmos desenvolvidos neste trabalho utiliza um método similar ao descrito acima para o mapeamento das tarefas, mas adota outros critérios para a seleção dos processadores, baseados na substituição de processadores e no uso de outra função de custo.

O algoritmo apresentado por Chatha e Verumi [5] realiza a síntese em *pipeline* de aplicações de processamento de sinais transformativos e busca otimizar o intervalo de iniciação do *pipeline* [18], minimizando o número de estágios do *pipeline* e a quantidade de memória requerida para passar os dados entre os estágios do *pipeline*. O algoritmo trabalha com combinações destas restrições podendo ser selecionadas ambas as restrições, somente uma ou mesmo nenhuma delas. No último caso, o algoritmo busca uma solução que otimize o intervalo de iniciação. A arquitetura dos sistemas consiste de um processador e um co-processador, nos quais as tarefas são mapeadas. A memória é

constituída de uma memória compartilhada usada na comunicação entre *hardware* e *software* e uma memória local usada na comunicação entre os módulos em *software*.

O algoritmo realiza o particionamento e o escalonamento das tarefas iterativamente para obter um particionamento que satisfaça as restrições de tempo e área. O particionamento usa uma abordagem *branch-and-bound* com uma única função objetivo que minimiza o intervalo de iniciação do sistema final. A cada nível do espaço de busca, o algoritmo seleciona uma tarefa não mapeada é selecionada e seu mapeamento é determinado. O desempenho de um determinado mapeamento é testado obtendo seu escalonamento em *pipeline*, através de um processo iterativo de escalonamento e transformações de retemporização. O escalonamento é baseado numa lista de tarefas prontas.

As transformações de retemporização são aplicadas quando o grafo não pode ser escalonado para um determinado valor de intervalo de iniciação e consistem da criação de um novo grafo com tarefas pertencentes a diferentes iterações do grafo original.

Este trabalho foca na otimização do intervalo de iniciação do sistema e na minimização de algumas métricas dos sistemas (latência da aplicação e quantidade de memória para passar os dados entre os estágios). Os algoritmos deste trabalho não tratam a otimização do intervalo de iniciação, mas a minimização de parâmetros de projeto.

Em [6], Dave, Lakshminarayana e Jha descrevem o COSYN, um algoritmo de síntese de sistemas embutidos que realiza o mapeamento das tarefas de um sistema, representado por um grafo de fluxo de dados, em um conjunto heterogêneo de processadores e elementos de hardware (ASIC e FPGA) e o escalonamento das tarefas atendendo restrições de tempo-real das aplicações. Para reduzir a complexidade, o COSYN realiza o agrupamento das tarefas para depois mapear os grupos nos processadores.

Para cada processador disponível para alocação das tarefas, seu custo e consumo médio e de pico de energia devem ser especificados, bem como arquitetura de memória, características do *link* de comunicação e características da memória *cache*.Para os ASICs deve ser especificado o custo e os atributos do empacotamento, como pinos e portas disponíveis e dissipação de calor média e de pico por porta. Similarmente, para o FPGA os seguintes atributos devem ser especificados, custo,

consumo de energia médio e de pico e atributos do empacotamento, como pinos disponíveis, número máximo de flip-flops ou de blocos de lógica combinacional ou de unidade funcionais programáveis.

Na primeira etapa do algoritmo, as principais estruturas de dados usadas pelas outras etapas do algoritmo são geradas. O algoritmo ordena as tarefas em ordem decrescente de prioridade que é calculada com base na distância das tarefas que é uma indicação do caminho mais longo no grafo partindo da tarefa e indo até a última tarefa de uma determinada linha de execução que contenha a tarefa.

A próxima etapa é a formação dos grupos de tarefas começando pela tarefa de maior prioridade. As tarefas de um grupo são interligadas pelas arestas do grafo numa linha de execução, isto é, a primeira tarefa é predecessora da segunda que é predecessora da terceira e assim até a última tarefa. Os grupos têm tamanho limitado pelo valor máximo para a soma dos tempos de execução das tarefas.

As próximas etapas do algoritmo são a alocação dos grupos nos processadores e o escalonamento das tarefas. Para cada grupo é montada uma lista com suas possíveis alocações e, aplicando um processo iterativo, o algoritmo determina qual processador da lista gera o melhor escalonamento para o grupo, a partir de uma avaliação de desempenho do escalonamento.

O agrupamento de tarefas de modo similar ao apresentado acima é usado no algoritmo de minimização do número de estágios com o objetivo de reduzir a latência do sistema.

No trabalho de Liu e Wong [15], o escalonamento tem o objetivo de minimizar a latência total do sistema com as tarefas sendo mapeadas inicialmente em *software* e posteriormente são movidas para *hardware*. Com relação à implementação em *hardware*, cada tarefa pode ter diferentes implementações em tecnologias diferentes, cada qual com seu custo e tempo de execução. As tarefas alocadas nos processadores e que não apresentam nenhuma dependência em relação a outras tarefas alocadas nos intervalos imediatamente anteriores são movidas para *hardware* e escalonadas num intervalo de tempo anterior ao que lhe foi designado no mapeamento original. Análise similar é feita com relação às tarefas das quais não dependem as tarefas alocadas nos intervalos imediatamente posteriores. A realocação das tarefas e a redução no tempo de execução das tarefas movidas para hardware visam atingir o objetivo de minimização da latência total da aplicação.

O algoritmo considera um mapeamento inicial de tarefas distribuindo o tempo de execução igualmente entre dois processadores iguais e escalonando as tarefas. A partir deste mapeamento, o algoritmo seleciona um número finito de tarefas para serem implementadas em *hardware* com o objetivo de minimizar a latência do sistema. O número de tarefas depende dos recursos de hardware (ASIC ou FPGA) disponíveis para o sistema. O critério para seleção das tarefas que serão implementadas em *hardware* considera as relações de dependência entre elas. Após o escalonamento inicial, as tarefas são classificadas como *forward* (F), *backward* (B) ou *stable* (X), segundo o conceito de mobilidade relativa da seguinte forma:

(1) *forward* (F) *task*: se uma tarefa τ é escalonada num intervalo de tempo t e não depende das tarefas τ_1 e τ_2 que estão escalonadas no intervalo de tempo t-1, a tarefa é classificada como *forward*. Neste caso, a tarefa τ pode ser movida para hardware para executar em paralelo com as tarefas em *software* que executam no intervalo t-1.

(2) *backward* (B) *task*: se uma tarefa τ é escalonada num intervalo de tempo t e as tarefas τ_1 e τ_2 que estão escalonadas no intervalo de tempo t+1 não dependem dela, a tarefa é classificada como *backward*. Neste caso, a tarefa τ pode ser movida para hardware para executar em paralelo com as tarefas em *software* que executam no intervalo t+1.

(3) *Stable* (X) *task*: Uma tarefa τ é definida como stable se ela é escalonada no tempo t e exista uma tarefa τ1 escalonada no intervalo de tempo t-1 e uma tarefa τ_2 escalonada no intervalo de tempo t+1, tal que a tarefa τ dependa de τ1 e a tarefa τ2 dependa da tarefa τ. Neste caso, a tarefa τ não pode ser movida sem que as outras tarefas ao seu redor também sejam movidas.

Após a classificação das tarefas, o algoritmo seleciona as tarefas que serão movidas para hardware, sempre buscando a minimização do tempo de execução e do custo do sistema e reescalona as tarefas do grafo.

Estes passos são repetidos até que todos os recursos de *hardware* tenham sido utilizados ou não haja mais tarefas para serem movidas para *hardware*.

Assim como no algoritmo descrito acima, nossos algoritmos priorizam a alocação das tarefas em software. Com a diferença que, neste algoritmo, a alocação em hardware é feita para minimizar a

latência do sistema e nos algoritmos desenvolvidos a tarefa é alocada em hardware quando a alocação em software não for possível.

O algoritmo descrito por Palazzari, Baldini e Coli em [16] realiza a síntese de *pipeline* para sistemas com tarefas periódicas e aperiódicas com o objetivo de satisfazer as restrições de tempo tanto das tarefas periódicas, como das tarefas aperiódicas, minimizando o *hardware* necessário para o sistema. As tarefas aperiódicas são caracterizadas por terem requisições aleatórias para suas execuções.

A idéia do algoritmo é realizar o mapeamento de um grafo global incluindo as tarefas periódicas e aperiódicas, usando um algoritmo *Simulated Annealing* [9]. Para a alocação das tarefas aperiódicas é adotada a premissa que a execução destas tarefas deve ser sincronizada com a execução das tarefas periódicas, assim qualquer requisição é executada no início do estágio do *pipeline* e as tarefas aperiódicas são mutuamente exclusivas, ou seja, somente uma requisição para execução das tarefas pode ser atendida entre duas ativações sucessivas das tarefas periódicas. A abordagem adotada para satisfazer as restrições de todas as tarefas é baseada no superdimensionamento do sistema com respeito ao que é necessário para satisfazer as tarefas periódicas, deste modo o pipeline é executado numa freqüência maior que a requisitada por estas tarefas.

O algoritmo determina inicialmente a quantidade de recursos de *hardware* necessária para alocar as tarefas do sistema, a seguir é determinado um primeiro mapeamento das tarefas nestes recursos e, a partir de então, é aplicado o algoritmo *Simulated Annealing* para minimizar os recursos. O algoritmo de minimização sucessivamente busca um mapeamento que respeite todas as restrições das tarefas, minimiza a área de silício e, se possível, a latência total do *pipeline*.

Neste trabalho, o mapeamento das tarefas e o escalonamento são formulados como um problema de minimização e um algoritmo *Simulated Annealing* é aplicado, enquanto que os algoritmos desenvolvidos neste trabalho utilizam métodos heurísticos com diferentes critérios de minimização.

Por último, o trabalho de Ranaweera e Agrawal em [17] descreve um algoritmo para o mapeamento de tarefas de uma aplicação periódica com prazos rígidos de término em um conjunto de processadores. O algoritmo utiliza um grafo de fluxo de dados para a representação das tarefas da aplicação e utiliza um conjunto de processadores onde as tarefas podem ser executadas. A escolha

dos processadores onde as tarefas serão alocadas ocorre durante a execução do algoritmo. Este algoritmo recorre a definição de grupos e a duplicação de tarefas para obter o mapeamento das tarefas agrupadas num *pipeline* de processadores que minimizam a latência total sistema ou o número de estágios. A latência do estágio não é uma restrição de entrada, sendo obtida como resultado deste algoritmo de síntese. Os grupos são formados com a finalidade de reduzir o tempo de comunicação agrupando num mesmo processador tarefas que trocam informações. A duplicação de tarefas também tem o objetivo de diminuir o tempo de comunicação entre tarefas que também trocam informações e que estão em grupos distintos.

O algoritmo consiste de cinco etapas principais. Na primeira etapa, o grafo é percorrido no sentido das tarefas inicias até as tarefas finais, ou seja, no sentido *top-down* e na etapa seguinte no sentido oposto, *bottom-up*. Nestas duas etapas, é calculado um conjunto de valores para cada tarefa, *Earliest Start Time* (est), *Earliest Completion Time* (ect), *Favorite Predecessor of each task* (fpred), *Favorite processors* (fprocs), *Latest Allowable Start Time* (last) e *Latest Allowable Completion Time* (lact), que são posteriormente usados na próxima etapa do algoritmo para a geração do conjunto inicial de grupos de tarefas usando uma quantidade pequena de processadores. O algoritmo trabalha com uma lista de tarefas prontas ordenada pelo nível da tarefa que é o maior valor para a soma dos tempos de computação das tarefas dos diferentes caminhos da tarefa até o final do grafo. A geração dos grupos é iniciada pelas tarefas finais e novas tarefas são agregadas aos grupos recursivamente até que o nó inicial seja encontrado.

O número de grupos gerados independe do número de processadores disponíveis. Assim, a quarta etapa envolve a duplicação de tarefas ou a compactação dos grupos. Se o número de grupos for menor que o número de processadores, ocorre a duplicação de tarefas e se o número de grupos for maior é feito o reagrupamento das tarefas para acomodar os grupos nos processadores disponíveis.

A última etapa do algoritmo é a geração do *pipeline* que consiste em alocar as tarefas nos estágios do *pipeline*.

Neste algoritmo, assim como em um dos algoritmos deste trabalho, as tarefas são agrupadas para diminuição dos tempos de comunicação. Entretanto, a latência do sistema é um parâmetro de saída

do algoritmo, ao contrário dos algoritmos deste trabalho que tem a latência como entrada e é usada no escalonamento das tarefas.

4.3 Conclusão

Neste capítulo apresentamos comentários sobre metodologias e algoritmos desenvolvidos para a síntese de sistemas digitais embutidos.

Pudemos notar que os algoritmos desenvolvidos adotaram diferentes modelos para a especificação dos sistemas, sendo que a escolha do modelo era decorrente das necessidades específicas de informação e da técnica escolhida para a síntese.

Alguns algoritmos foram desenvolvidos com o objetivo de atender a classes específicas de aplicação, como, por exemplo, as aplicações transformativas descritas por Chatha e Verumi em [5], e as aplicações compostas de tarefas periódicas e aperiódicas descritas por Palazzari, Baldini e Coli em [16]. Pode-se notar, também, que diferentes estruturas de execução das tarefas, descritas no capítulo anterior, foram adotadas.

No capítulo seguinte são descritos os algoritmos desenvolvidos neste trabalho para a síntese de sistemas digitais heterogêneos compostos por diferentes processadores e elementos dedicados (*hardware*). Os algoritmos adotam uma estrutura em *pipeline* de processadores para a execução das tarefas atendendo à restrição de desempenho do sistema ditada pela taxa máxima de chegada dos dados, ao mesmo tempo em que procuram otimizar diferentes parâmetros de qualidade dos sistemas.

Capítulo 5

Algoritmos

Neste capítulo serão descritos os algoritmos para síntese de sistemas embutidos desenvolvidos como parte deste trabalho, bem como as informações de entrada e saída.

As informações de entrada usadas pelos algoritmos incluem a especificação dos sistemas e a biblioteca de componentes. Os algoritmos realizam a síntese de um *pipeline* para o sistema, apresentando como saídas a alocação de processadores, o mapeamento das tarefas e o escalonamento, além do custo, do número de processadores e do número de estágios usados para comparar várias implementações.

5.1 Introdução

O projeto de um sistema embutido leva em conta diferentes fatores que influenciam na escolha da metodologia a ser adotada para sua síntese. Entre estes fatores podemos citar as restrições da aplicação, tais como: tempo de execução, desempenho do sistema, custo da solução ou restrições de área e de consumo de energia. Os algoritmos desenvolvidos neste trabalho têm como objetivo realizar a síntese de sistemas embutidos atendendo à restrição de desempenho dos sistemas representada pela taxa de chegada dos dados, através de uma estrutura de *pipeline* de processadores para a execução das tarefas. Além do atendimento a esta restrição de desempenho, cada algoritmo procura minimizar um parâmetro específico do projeto.

Foram desenvolvidos três algoritmos que têm os seguintes objetivos:

1) Minimização do número de processadores do sistema (MPS) através da substituição de processadores do sistema como uma alternativa a alocação de novos processadores para completar a síntese;
2) Minimização do custo do sistema (MCS) através da alocação de processadores que possam apresentar o menor custo para execução das tarefas;
3) Minimização da latência total do sistema (MLS) por meio da redução do número de estágios do pipeline.

As seções seguintes descrevem os parâmetros de entrada do algoritmo, que são a especificação do sistema através de um modelo, a taxa máxima de recepção dos dados e a biblioteca de componentes, e os algoritmos desenvolvidos.

5.2 Entradas dos algoritmos

Os algoritmos implementados utilizam, como entrada, o seguinte conjunto de informações:

a) Especificação do sistema: representada por um grafo de fluxo de dados, mostrando as tarefas do sistema e suas interdependências e pela taxa máxima de recepção de dados que depende da aplicação do sistema;
b) Biblioteca de Componentes com informações sobre os processadores que podem executar as funções do sistema;

Os algoritmos utilizam um *pipeline* de processadores para execução das tarefas e também fornecem um conjunto de métricas para a avaliação da síntese. As métricas incluem informações sobre utilização dos processadores, custo da implementação e latência total da implementação.

5.2.1 Especificação do Sistema

Os algoritmos trabalham com a especificação funcional dos sistemas representada por grafos de fluxo de dados (GFD), onde os nós representam as tarefas do sistema e as arestas representam as

dependências de dados entre as tarefas. A Figura 5.1 mostra um exemplo de um grafo com 12 tarefas, nomeadas de T1 a T12. A representação do grafo contém dois nós adicionais de controle, representados em cinza no desenho, que não representam tarefas do sistema. Estes nós estão incluídos no grafo para indicar, nos caminhos do grafo, quais são as tarefas iniciais e finais de cada um. Todas as tarefas iniciais do grafo, no exemplo, T1, T6 e T9, são conectadas ao primeiro nó, e as tarefas terminais, T3, T5, T8 e T12, são conectadas ao último nó.

Os nós do grafo contêm informações sobre as funções desempenhadas pelas tarefas do sistema. O tempo de execução destas tarefas, entretanto, não aparece no grafo, pois depende dos processadores alocados durante a síntese. As arestas representam os relacionamento entre as tarefas do grafo, contendo informações sobre o fluxo dos dados dentro do sistema. Uma tarefa Ti é antecessora de outra Tj se houver uma aresta no grafo saindo de Ti e chegando em Tj. Uma tarefa somente pode ser executada após todas suas tarefas antecessoras terem sido executadas e enviados os dados. No exemplo da Figura 5.1, a tarefa T1 é antecessora das tarefas T2 e T4, a tarefa T6 é antecessora de T7 e T9 é antecessora das tarefas T10 e T11 que, por sua vez, são antecessoras da tarefa T12.

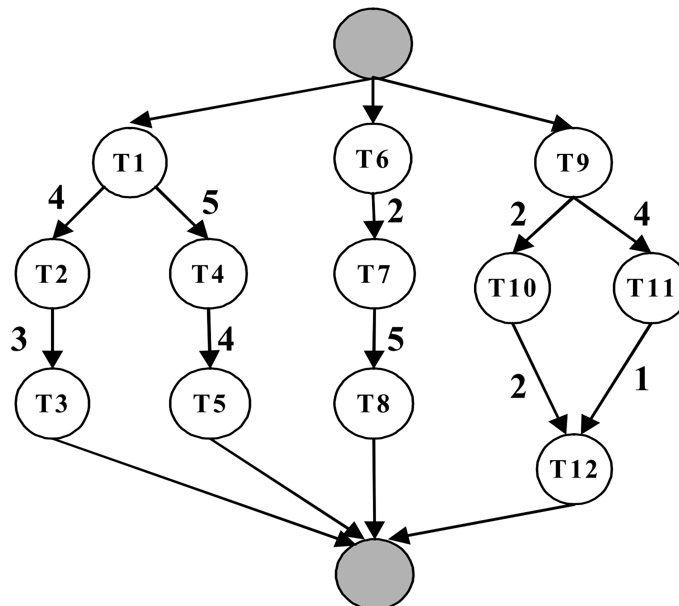


Figura 5.1 - Grafo de tarefas

As arestas do grafo apresentam um valor associado que é o tempo necessário para a transferência dos dados entre as tarefas. O tempo de transferência dos dados entre as tarefas T1 e T2 é de 4 unidades de tempo e entre T1 e T4 é de 5 unidades de tempo. O tempo de transferência depende da quantidade de dados a ser transferida entre as tarefas e da taxa de transferência do canal utilizado para realizar a comunicação. Este trabalho não tem o objetivo de realizar o projeto dos canais de comunicação. Desta forma, o tempo de transferência dos dados é fornecido para cada aresta do grafo, de acordo com um meio de comunicação previamente escolhido.

Assim, o início da execução de uma tarefa que tenha uma ou mais tarefas antecessoras depende do tempo de execução destas tarefas e do tempo de transferência dos dados. Este início só poderá se dar após o término da execução de todas as tarefas antecessoras e o envio dos respectivos dados à tarefa. Entretanto, o instante exato de início da execução da tarefa somente é definido no escalonamento, pois dependerá da disponibilidade do processador onde a tarefa está alocada.

5.2.2 Taxa de recepção dos dados

A taxa de recepção dos dados é um importante parâmetro de entrada dos algoritmos e define o intervalo entre cada chegada dos conjuntos de dados a serem processados no sistema. Esta taxa é a restrição de tempo do sistema sendo utilizada em todas as etapas dos algoritmos, desde o particionamento das tarefas até o escalonamento final do sistema. O intervalo entre cada chegada dos dados, que é o inverso da taxa de recepção, define a latência do estágio do *pipeline* e é usado como parâmetro para limitar o tempo máximo de execução das tarefas do sistema, pois uma restrição imposta pelos algoritmos é que as tarefas devem iniciar e finalizar sua execução dentro de um único estágio.

5.2.3 Biblioteca de Componentes

Como anteriormente mencionado, o resultado dos algoritmos é um sistema sintetizado que consiste de um ou mais processadores programáveis e dedicados (*hardware)*. Os processadores

programáveis são compartilhados por um conjunto de componentes de *software*. Um componente de *software* é uma implementação de uma tarefa do sistema que pode ser executada num processador, enquanto que um componente de *hardware* é um processador dedicado que executa uma função. Para selecionar a melhor implementação para uma determinada tarefa do sistema cada algoritmo de síntese adota critérios específicos, dependendo dos parâmetros que se pretende minimizar, explorando as diferentes características dos processadores, programáveis e dedicados.

Os componentes disponíveis para a síntese são armazenados em uma biblioteca de projeto, denominada Biblioteca de Componentes. Um componente de *software,* ou implementação em *software*, é descrito pelas seguintes informações:

- Nome da tarefa que ele implementa;
- Tempo de execução;
- Identificação do processador onde ele é executado.

Um componente de *hardware*, ou implementação em *hardware*, é descrito por:

- Nome da tarefa que ele implementa;
- Tempo de execução;

Por exemplo, a Tabela 5-1 mostra a Biblioteca de Componentes usada na síntese do sistema representado na Figura 5.1. Nesta biblioteca, as tarefas T1 a T12 têm implementação em *software* para 3 diferentes processadores, P1, P2 e P3. Para a tarefa T10, existe uma implementação no processador dedicado denominado HW1.

De acordo com a função de custo adotado neste trabalho, o custo da tarefa para as implementações em *software* é o custo do processador. Para as implementações em *hardware*, o custo da tarefa é o custo dos recursos de *hardware* necessários para implementar a tarefa na tecnologia definida pelo projetista. Os custos dos processadores e dos recursos de *hardware* são também armazenados na Biblioteca de Componentes. O custo do projeto dependerá da quantidade de processadores e de recursos de hardware alocados. Uma premissa dos algoritmos é que quanto mais rápido for um processador maior seu custo [1], assim o tempo de execução de uma tarefa decresce à medida que um processador mais caro é selecionado para implementá-la.

Tabela 5-1 – Informações da Biblioteca de Componentes

Tarefas	Processadores			HW
	P1	*P2*	*P3*	*HW1*
T1	6	5	3	
T2	15	13	11	
T3	5	3	3	
T4	7	5	4	
T5	8	7	4	
T6	6	5	4	
T7	8	5	4	
T8	5	4	2	
T9	5	3	2	
T10	90	70	60	10
T11	5	4	3	
T12	7	4	2	

Os componentes da biblioteca que implementam as funções do sistema são identificados pelos algoritmos no início do processo de síntese. Somente os componentes cujo tempo de execução não excedem a latência do estágio são considerados pelos algoritmos. Assim, para diferentes valores de taxa de recepção dos dados os componentes que são candidatos a implementar as funções do sistema podem variar.

5.3 Alocação, Mapeamento e Escalonamento em Estruturas Pipeline

Os algoritmos para síntese realizam o particionamento *hardware-software*, a alocação dos processadores, o mapeamento e o escalonamento das tarefas do sistema, seguindo as etapas mostradas na Figura 5.2. Como a alocação de processadores e o escalonamento de tarefas são problemas classificados como NP-Completos [10], muitos dos algoritmos desenvolvidos aplicam

métodos heurísticos para suas resoluções [1] [2] [6] [15] [17] [21]. Esta abordagem é seguida também neste trabalho.

Nas próximas seções as etapas do processo de síntese serão descritas detalhadamente. As primeiras seções descrevem as etapas de particionamento *hardware-software*, cálculo das distâncias e escalonamento. Estas etapas são realizadas da mesma forma no início de todos os algoritmos.

5.3.1 Particionamento *hardware-software*

A primeira etapa dos algoritmos é o particionamento *hardware-software* das tarefas que consiste em determinar a forma de implementação para cada tarefa do sistema. A forma de implementação de uma tarefa define se ela será executada em um processador programável ou em um processador dedicado.

Na seleção da implementação de uma tarefa são consideradas todas as implementações disponíveis na Biblioteca de Componentes cujo tempo de execução não exceda a latência do estágio. Os algoritmos priorizam a alocação das tarefas em processadores programáveis, deixando em processadores dedicados, somente as tarefas que não tenham uma implementação em *software* cujo tempo de execução não exceda a latência do estágio definida para o sistema. A priorização do uso dos processadores programáveis em relação aos elementos de *hardware* ocorre porque os processadores podem ser compartilhados entre várias tarefas ao contrário do *hardware* que é alocado a tarefas específicas. Esta flexibilidade no uso dos processadores programáveis pode propiciar uma melhor relação custo versus benefício, principalmente quando se está considerando um novo projeto onde os elementos de processamento ainda não foram definidos.

Portanto, o particionamento das tarefas depende da Biblioteca de Componentes e da latência do estágio. É possível que, para uma dada configuração de entrada, não haja implementação para todas as tarefas. Por exemplo, para a Biblioteca de Componentes representada na Tabela 5-1, a tarefa T10 não tem implementação possível se a latência do estágio for 5 unidades de tempo, tem implementação somente em *hardware* para estágio com latência de 10 unidades de tempo e

finalmente tem implementação em *software* para estágios com latência maior ou igual a 60 unidades de tempo.

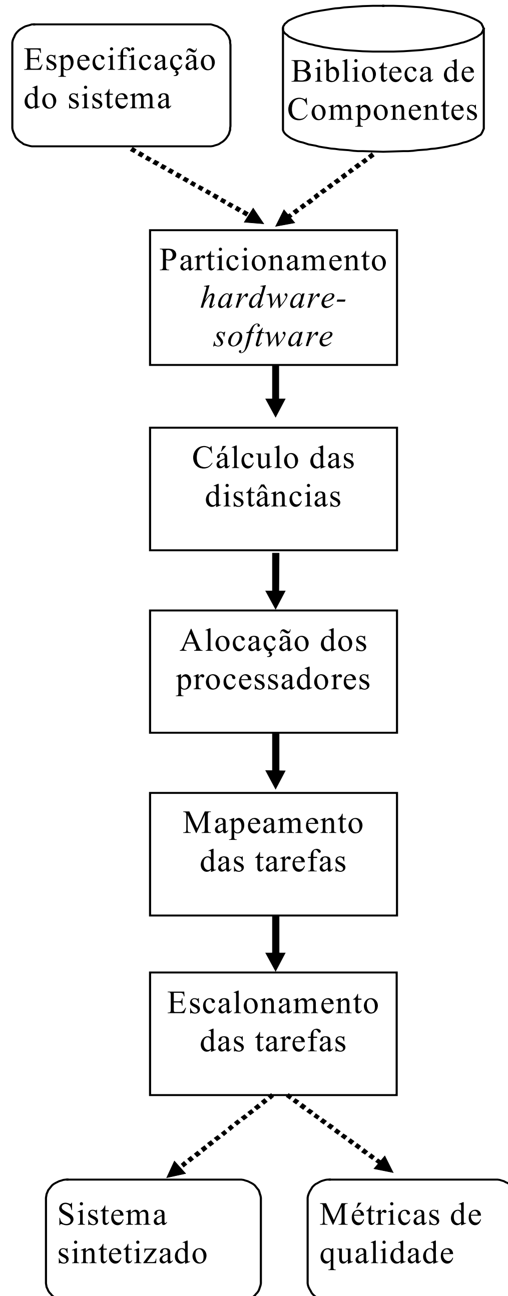


Figura 5.2 - Etapas dos algoritmos de síntese de sistemas embutidos

5.3.2 Cálculo das distâncias

A próxima etapa do algoritmo é calcular a distância das tarefas. Os algoritmos realizam a alocação, o mapeamento e o escalonamento tarefa por tarefa segundo uma lista de tarefas, começando pela tarefa com a maior distância e seguindo a ordem decrescente de distância.

A distância de uma tarefa é definida como o tempo máximo transcorrido entre o início da execução da tarefa e o término da execução da última tarefa num caminho do grafo a partir da tarefa, considerando para o cálculo os tempos de execução das tarefas e de comunicação entre as tarefas. O cálculo da distância assume o pior caso para a execução das tarefas do sistema, ou seja, é usado o maior tempo de execução para as tarefas considerando todas as implementações na Biblioteca de Componentes que satisfaçam a restrição da latência do estágio. O tempo de comunicação entre as tarefas também é considerado no cálculo, uma vez que o pior caso ocorre se as tarefas forem alocadas em processadores distintos. A Figura 5.3 mostra os dois caminhos possíveis para a tarefa T1, a partir dos quais sua distância será calculada e a Tabela 5-2 apresenta o cálculo das distâncias para o grafo da Figura 5.1, usando a Biblioteca de Componentes da Tabela 5-1 para estágios com latência de 50 unidades de tempo.

O cálculo das distâncias das tarefas é feito percorrendo o grafo de "baixo para cima" iniciando pelas tarefas terminais, denominadas terminais. No grafo apresentado, as tarefas T3, T5, T8 e T12 são tarefas terminais. Para estas tarefas, a distância é igual ao tempo de execução da implementação mais lenta para a tarefa disponível na Biblioteca de Componentes. Conforme mostra a tabela, as distâncias associadas a essas tarefas são 5, 8 e 7 respectivamente.

O cálculo das distâncias prossegue com tarefas cujas sucessoras tem as distâncias já calculadas. No grafo, são as tarefas T2, T4, T7, T10 e T11. A distância é calculada como a soma do tempo de execução da tarefa, considerando a implementação mais lenta, com o caminho mais lento a partir da tarefa, incluindo o tempo de comunicação entre a tarefa e sua sucessora neste caminho. A distância da tarefa T2 é calculada pela expressão **Max (15, 13, 11) + 3 + 5**, onde a função **Max** retorna o maior valor entre os valores passados como parâmetro. A expressão, **Max (15, 13, 11),** determina o maior tempo de execução para a tarefa T2, a partir das implementações na Biblioteca de

Componentes. Os próximos termos da expressão são, respectivamente o tempo de comunicação entre T2 e T3 e a distância da tarefa T3.

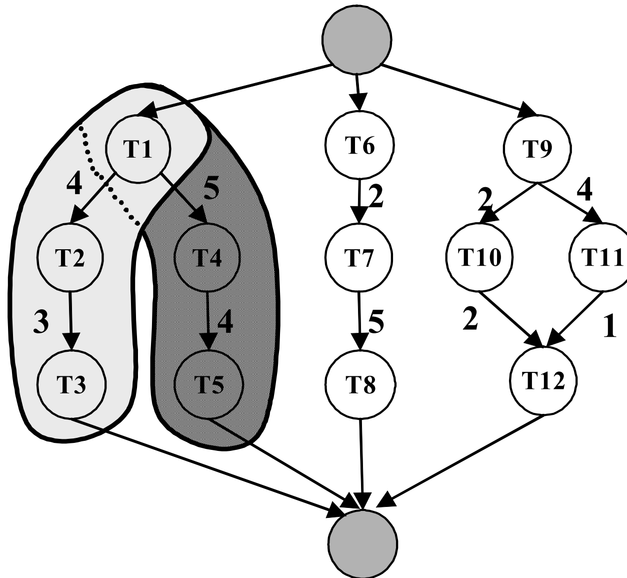


Figura 5.3 – Caminho do grafo a partir da tarefa T1

Para a tarefa T1, a distância é calculada por **Max (6, 5, 3) + Max (4 + 23, 5 + 19)**, que é igual a 33. Aqui a primeira parte da expressão, **Max (6, 5, 3)**, determina o maior tempo de execução para a tarefa T1, e a segunda parte da expressão, **Max (4 + 23, 5 + 19)** , determina, entre os dois caminhos do grafo a partir de T1, o mais lento. O primeiro parâmetro nesta função apresenta o tempo de comunicação entre T1 e T2, e a distância de T2 e o segundo parâmetro representa as mesmas informações para o caminho que contém a tarefa T4.

A ordem das tarefas na lista de tarefas, em ordem decrescente de distância, é T1, T6, T9, T2, T4, T10, T7, T11, T5, T12, T3 e T8.

Tabela 5-2 - Cálculo da distância das tarefas

Tarefa	Fórmula de cálculo	Distância
T1	Max (6, 5, 3) + Max (4 + 23, 5 + 19)	33
T2	Max (15, 13, 11) + 3 + 5	23
T3	Max (5, 3, 3)	5
T4	Max (7, 5, 4) + 4 + 8	19
T5	Max (8, 5, 4)	8
T6	Max (6, 5, 4) + 2 + 18	26
T7	Max (8, 5, 4) + 5 + 5	18
T8	Max (5, 4, 2)	5
T9	Max (5, 3, 2) + Max (2 + 19, 4 + 13)	26
T10	10 + 2 + 7	19
T11	Max (5, 4, 3) + 1 + 7	13
T12	Max (7, 4, 2)	7

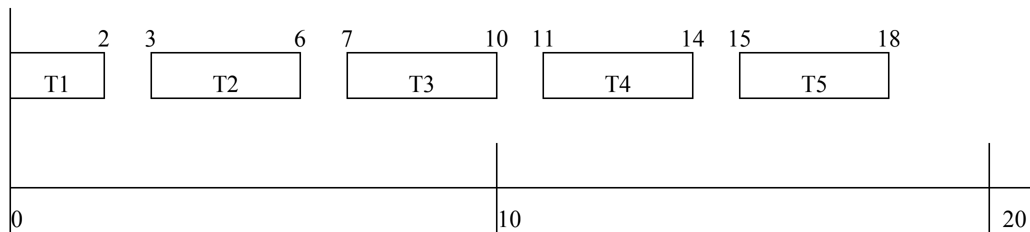
5.3.3 Escalonamento

Como descrito anteriormente, o escalonamento das tarefas é feito considerando a organização em *pipeline* e seguindo a restrição dos algoritmos de que uma tarefa deve iniciar e terminar sua execução no mesmo estágio. Uma tarefa é escalonada levando em conta o processador em que ela foi mapeada na fase de mapeamento. Um processador pode executar tarefas em diferentes estágios, desde que a soma total dos tempos de execução não exceda o tempo de latência do estágio e as tarefas não se sobreponham no tempo, pois a execução das tarefas nos estágios é concorrente.

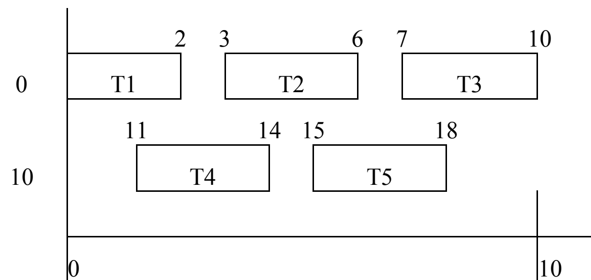
Na Figura 5.4 podemos ver um exemplo de sobreposição de tarefas. Todas as tarefas foram alocadas em um mesmo processador, e o estágio tem latência de 10 unidades de tempo. Até o instante 10, somente as tarefas do primeiro estágio estão em execução. A partir deste instante, as tarefas do segundo estágio serão executadas em paralelo com as tarefas do primeiro estágio que estarão processando um novo conjunto de dados, como podemos ver na Figura 5.4 (b).

Antes do escalonamento de uma tarefa é necessário determinar o instante mais cedo em que a tarefa pode ser iniciada. Este instante, chamado de *earliest start time* (EST) da tarefa, corresponde ao instante no qual a tarefa completa a recepção dos dados de suas tarefas antecessoras. Uma tarefa pode ser escalonada em um processador em qualquer instante a partir de seu EST. O instante exato vai depender da ocupação deste processador.

Devido à restrição que a tarefa deve iniciar e terminar seu processamento dentro do mesmo estágio, eventualmente, o início da tarefa precisa ser adiado, passando para o próximo estágio. O EST das tarefas não-escalonadas é calculado toda vez que uma tarefa do sistema for mapeada e escalonada.



(a) Distribuição das tarefas no pipeline



(b) Distribuição das tarefas no tempo

Figura 5.4 - Distribuição das tarefas no pipeline (a) e a distribuição no tempo (b)

5.3.4 Alocação, mapeamento e escalonamento das tarefas

Depois do cálculo das distâncias e da organização das tarefas em uma lista ordenada por distância decrescente, os algoritmos prosseguem fazendo a alocação de processadores, o mapeamento das tarefas nos processadores e o escalonamento para cada uma das tarefas na lista. Cada algoritmo sintetiza o sistema em vários passos. Quando uma tarefa é considerada, todas as tarefas de maior distância, incluindo suas antecessoras, já foram mapeadas em processadores e escalonadas. Em cada passo, um recurso anteriormente alocado ou um novo recurso é utilizado para mapeamento e escalonamento da tarefa correspondente a este passo.

Conforme descrito nas seções a seguir, cada algoritmo adota procedimentos distintos para alocação de processadores, mapeamento e escalonamento de tarefas.

5.3.4.1 Síntese de Pipeline com Minimização dos Processadores do Sistema (MPS)

Os objetivos do algoritmo de Síntese de Pipeline com Minimização dos Processadores do Sistema (MPS) são o atendimento da restrição de desempenho do sistema e a minimização do número de processadores no sistema sintetizado. Um número pequeno de processadores facilita o projeto das estruturas de conexão. Além de minimizar o número de processadores, o algoritmo tem como objetivo secundário a redução do custo do sistema.

Este algoritmo adota um processo iterativo de alocação e substituição de processadores, escolhendo a opção de menor custo para mapear e escalonar as tarefas. O algoritmo MPS executa os seguintes passos para a primeira tarefa Ti na lista de tarefas prontas:

1. Procura, entre os processadores anteriormente alocados, um no qual a tarefa Ti possa ser mapeada e escalonada. Se houver mais de um processador em que a tarefa possa ser escalonada, é selecionado aquele no qual a execução da tarefa Ti termina mais cedo;
2. Se nenhum processador for encontrado, tenta-se substituir um dos processadores alocados por outro mais veloz que permita escalonar todas as tarefas do processador a ser substituído e também a tarefa Ti. Para cada um dos processadores alocados, considera-se como candidatos

a substituí-lo, todos os processadores de maior custo na Biblioteca de Componentes. O critério de custo é adotado devido à premissa que o tempo de execução das tarefas diminui com o aumento do custo do processador.

Para reduzir o acréscimo do custo do sistema, considera-se os processadores em ordem crescente de custo. Logo que um processador capaz de executar todas as tarefas do processador mais a tarefa Ti for encontrado, o processador original é substituído na relação de processadores alocados para o sistema.

3. O último passo executado, caso nenhum processador possa ser substituído no passo anterior, consiste na alocação de um novo processador para escalonar a tarefa. É selecionado o processador mais barato que possa alocar a tarefa dentro de um estágio. Esta escolha corresponde a implementação com o maior tempo de execução para a tarefa, mas traz um menor acréscimo ao custo do sistema.

O custo da substituição de um processador é igual a diferença entre os custos dos processadores. Esta estratégia, entretanto, pode não gerar uma solução otimizada em termos de custo total. Embora, o número de processadores tenda a ser minimizado, o custo total não necessariamente será minimizado, pois os processadores mais rápidos têm um custo maior.

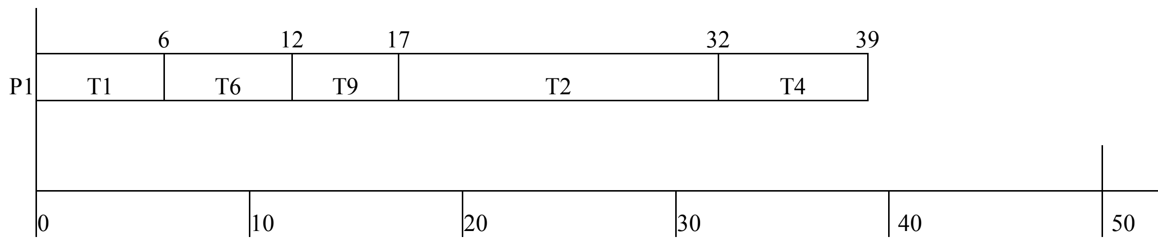
A Figura 5.5 mostra o escalonamento do grafo da Figura 5.1 com uma latência de estágio de 50 unidades de tempo usando a Biblioteca de Componentes apresentada na Tabela 5-1. Os custos dos processadores são: P1 – $2.00; P2 – $5.00; P3 – $14.00 e HW1 – $25.00.

De acordo com as distâncias calculadas na Tabela 5-2, a primeira tarefa a ser mapeada é T1. Como ainda não existe nenhum processador alocado, um novo processador do tipo mais barato é selecionado, neste caso do tipo P1. Este processador é usado posteriormente para mapear as quatro próximas tarefas da lista, que são T6, T9, T2 e T4, conforme mostra a Figura 5.5(a). A próxima tarefa da lista, T10, não pode ser mapeada num processador programável para a latência estabelecida, assim a tarefa é implementada pelo processador dedicado HW1, conforme mostrado na Figura 5.5(b). A próxima tarefa, T7, também é escalonada em P1, como mostra a Figura 5.5(c).

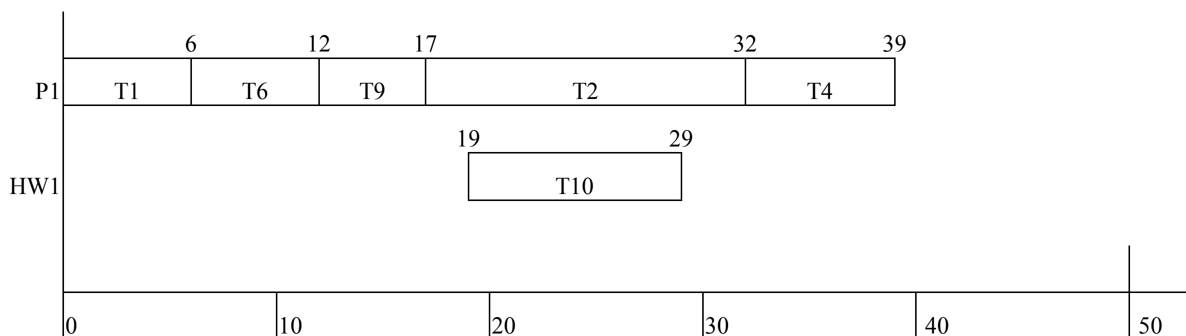
A tarefa T11 não pode ser mapeada em P1, pois o tempo total de execução neste processador ultrapassaria 50 unidades de tempo. É necessário, portanto, alocar um novo processador. Antes de

alocá-lo, verifica-se a possibilidade de substituir P1 por outro processador. Como todas as tarefas têm implementação em P2, é feita a substituição dos processadores e a tarefa T11 é escalonada, como mostra a Figura 5.5(d). Neste processador é possível escalonar, ainda, as tarefas seguintes da lista, T5 e T12, conforme pode ser visto na Figura 5.5(e).

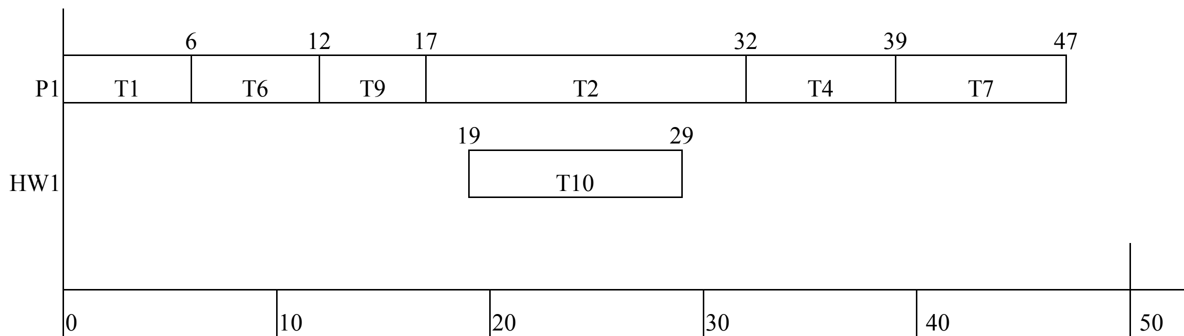
A tarefa T3 não pode ser escalonada em P2, pois o processador com o tempo total de execução das tarefas igual a 50 está completamente ocupado. Assim, é verificada a possibilidade da substituição por um outro processador. É selecionado um processador do tipo P3, que possui implementação para todas as tarefas de P2 e também para a tarefa T3, conforme pode ser visto na Figura 5.5(f). Esta configuração permite escalonar a última tarefa do grafo, T8. Finalmente, a Figura 5.5(g) mostra o escalonamento final do grafo.



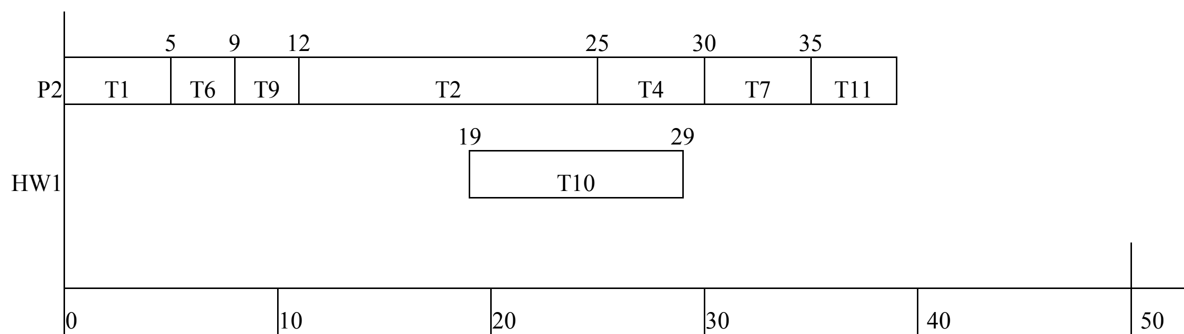
(a) Escalonamento das tarefas T1, T6, T9, T2 e T4 no processador P1



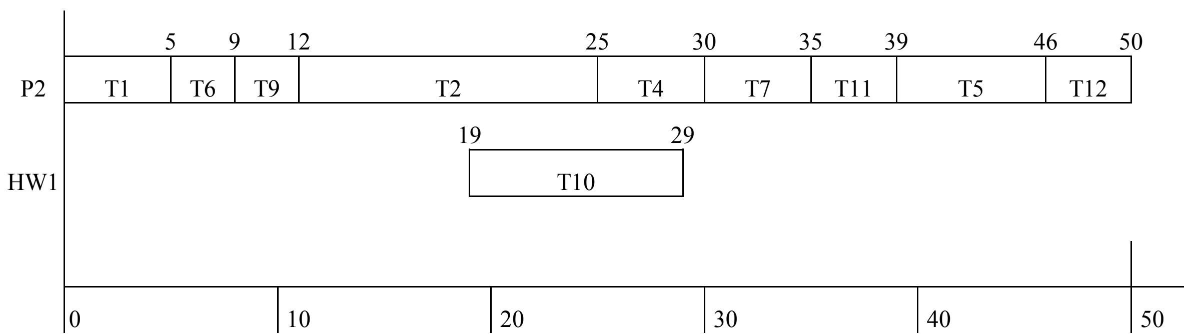
(b) Escalonamento da tarefa T10 em HW1



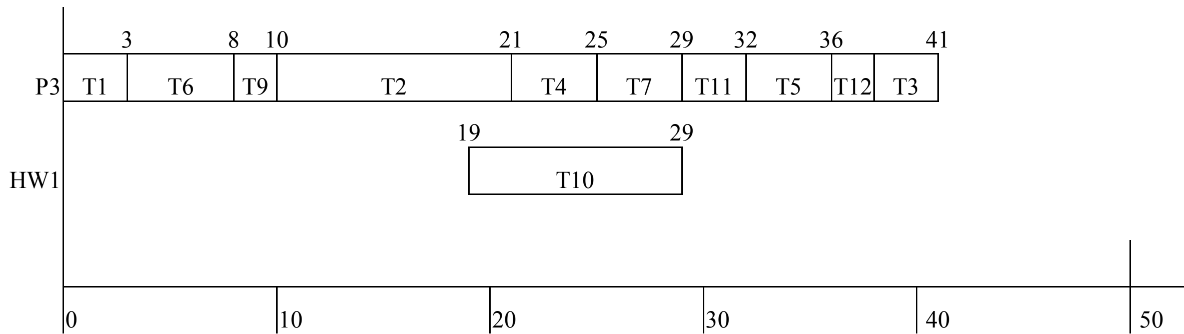
(c) Escalonamento da tarefa T7 em P1



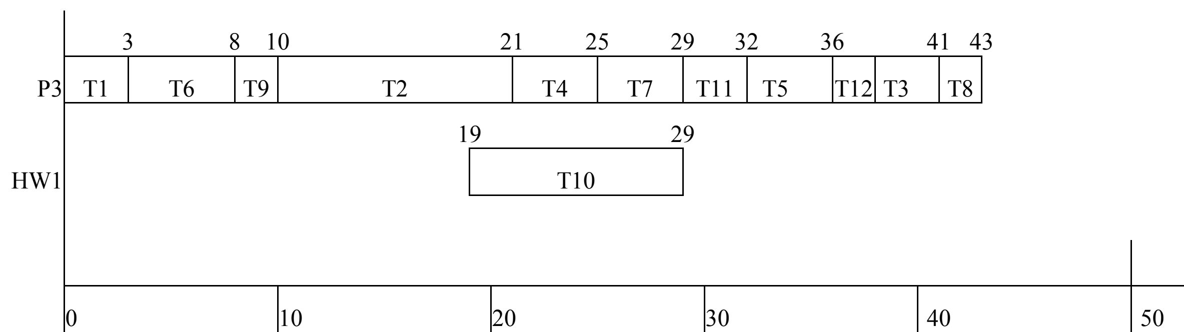
(d) Substituição do processador P1 por P2 para alocação da tarefa T11



(e) Escalonamento das tarefas T5 e T12 em P2



(f) Substituição do processador P2 por P3 para alocação da tarefa T3



(g) Escalonamento final das tarefas

Figura 5.5 – Escalonamento das tarefas aplicando o algoritmo MPS.

Abaixo são apresentadas as métricas para avaliação da qualidade da síntese do sistema produzida pelo algoritmo MPS. O custo final da solução é a soma dos custos dos processadores programáveis e do *hardware*.

Numero de processadores: 1

Custo: $39,00

Número de estágios: 1

Latência Total: 50

Utilização Média: 53%

5.3.4.2 Síntese de Pipeline com Minimização do Custo do Sistema (MCS)

O algoritmo de Síntese de Pipeline com Minimização do Custo do Sistema (MCS) tem como objetivos atender a restrição de desempenho do sistema e minimizar o custo total dos processadores do sistema, independente do número total de processadores alocados.

Este algoritmo aplica o conceito de Custo Remanescente (CR) para a alocação dos processadores do sistema. O cálculo do Custo Remanescente é baseado na estimativa simples do custo necessário para mapear as tarefas ainda não escalonadas em processadores de um único tipo. Assim, cada processador da Biblioteca de Componentes que implementa as tarefas do sistema tem um CR associado.

O cálculo do CR de um processador é baseado em seu Tempo Remanescente (TR) definido como a soma dos tempos de execução das tarefas não-escalonadas que podem ser mapeadas neste processador. O cálculo do TR, feito para encontrar um processador para a execução de T_i, é mostrado a seguir:

$$\mathrm{TR}(\mathrm{T}_i, P) = \sum^{n} \mathrm{TExec}(\mathrm{T}_J, P) \qquad \text{(eq. 5.1)}$$

onde:

$\mathrm{TExec}(\mathrm{T}_J, \mathrm{P})$ é o tempo de execução da tarefa não escalonada J no processador P.

Se não existe uma implementação para a tarefa T_J no processador ou se o tempo de execução exceder a latência do estágio, o tempo de execução, $\mathrm{TExec}(\mathrm{T}_J, \mathrm{P})$, é zero. O Custo Remanescente (CR) de um tipo de processador é o número de processadores deste tipo necessários para executar as tarefas não-escalonadas considerando a divisão das tarefas nos estágios do *pipeline*, multiplicado pelo custo do processador. O cálculo do CR é mostrado a seguir:

$$CR(T_i, P) = \left\lceil \frac{TR(T_i, P)}{Le} \right\rceil * Custo(P) \quad \text{(eq. 5.2)}$$

onde:

Custo(P) é o custo do processador *P*; e

Le é a latência do estágio do *pipeline*;

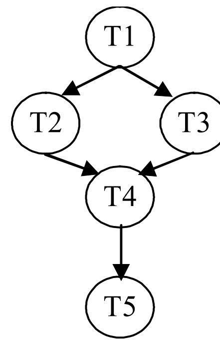
A hipótese de que um único tipo de processador será usado para completar a síntese simplifica a estimativa do impacto da escolha de um processador no custo final do sistema. Esta estimativa só pode ser usada se todos os processadores programáveis puderem ser usados para implementar todas as tarefas do sistema. Quando isto não acontece, é necessária uma estimativa diferente do custo remanescente. Vale observar que, apesar do uso do CR como uma estimativa simplificada do custo final do sistema, este algoritmo não leva à síntese de um sistema constituído por um único tipo de processador. Isto se deve a atualização do CR antes da alocação de um novo processador.

Após o mapeamento de uma tarefa, o tempo remanescente TR de cada processador é atualizado, subtraindo de seu valor, o tempo de execução da tarefa alocada no processador.

Assim como no algoritmo MPS, as etapas de alocação, mapeamento e escalonamento são repetidas para cada tarefa. No algoritmo MCS são descritos os passos a seguir para cada tarefa Ti:

1. Cálculo dos valores do $TR(T_i, P_k)$ e $CR(T_i, P_k)$ dos processadores;
2. Procura-se, entre os processadores alocados, um processador onde a tarefa possa ser mapeada e escalonada. Se houver pelo menos um que possa escalonar a tarefa, é selecionado aquele em que o instante de término da execução for mais cedo;
3. Se nenhum processador for encontrado no passo anterior, um novo processador P é alocado, sendo selecionado aquele com o menor custo remanescente $CR(T_i, P_k)$.

O exemplo apresentado na Figura 5.6, mostra o escalonamento das tarefas de um grafo de 5 tarefas aplicando-se o algoritmo MCS. O tempo de comunicação entre as tarefas é zero para simplificar o exemplo. A latência do estágio do *pipeline* é de 10 unidades de tempo e a Biblioteca de Componentes tem implementações para as tarefas em 2 processadores P1 e P2, cujos custos são, respectivamente, $ 8.00 e $5.00.



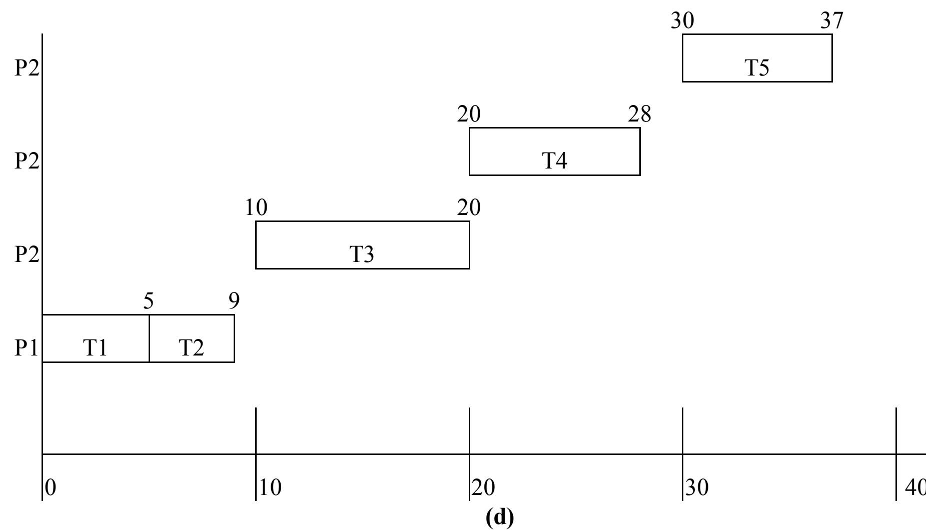
(a)

Tarefa	P1 ($8.00)	P2 ($5.00)
T1	5	8
T2	4	9
T3	8	10
T4	6	8
T5	5	7

(b)

Tarefa	Distância
T1	8 + 25 = 33
T2	9 + 15 = 24
T3	10 + 15 = 25
T4	8 + 7 = 15
T5	7

(c)



(d)

Figura 5.6 - Grafo de tarefas (a), Biblioteca de Componentes (b), cálculo das distâncias (c) e o escalonamento final do sistema (d)

A ordem das tarefas na lista de tarefas, seguindo a ordem decrescente das distâncias calculadas na Figura 5.6(c) é T1, T3, T2, T4 e T5. Na alocação de um processador par executar a primeira tarefa da lista, T1, o CR de cada processador é calculado e o processador P1 é selecionado, pois apresenta o CR menor, \$24.00, conforme é mostrado a seguir:

$TR(T1, P1) = 5 + 4 + 8 + 6 + 5 = 28$ $\quad$ $CR(T1, P1) = \lceil 28 / 10 \rceil * \$8.00 = \$24.00$

$TR(T1, P2) = 8 + 9 + 10 + 8 + 7 = 42$ $\quad$ $CR(T1, P2) = \lceil 42 / 10 \rceil * \$5.00 = \$25.00$

Para o mapeamento da próxima tarefa da lista, T3, um novo processador deve ser alocado, uma vez que o processador P1, anteriormente selecionado, não pode executá-la, pois só tem 5 unidades de tempo disponível. Primeiro os tempos remanescentes TR dos processadores são atualizados, conforme mostrado abaixo:

$TR(T3, P1) = 28 - 5 = 23$

$TR(T3, P2) = 42 – 8 = 34$

Para a escolha do novo processador os CRs dos processadores são atualizados, conforme visto abaixo, e um novo processador do tipo P2 é selecionado.

$CR(T3, P1) = \lceil 23 / 10 \rceil * \$8.00 = \$24.00$

$CR(T3, P2) = \lceil 34 / 10 \rceil * \$5.00 = \$20.00$

A próxima tarefa da lista, T2, pode ser mapeada em um dos processadores já alocados, logo somente é feita a atualização dos TRs:

$TR(T2, P1) = 23 - 8 = 15$

$TR(T3, P2) = 34 - 10 = 24$

Para o mapeamento da tarefa T4 também será necessário alocar um novo processador, porque não é possível mapeá-la nos processadores anteriormente alocados. Após a atualização dos TRs e CRs, conforme visto abaixo, novamente é selecionado um processador do tipo P2 que tem o menor CR.

TR(T4, P1) = 15 - 4 = 11 $\quad$ CR(T4, P1) = $\lceil 11 / 10 \rceil$ * \$8.00 = \$16.00

TR(T4, P2) = 24 - 9 = 15 $\quad$ CR(T4, P2) = $\lceil 15 / 10 \rceil$ * \$5.00 = \$10.00

Por último, para a tarefa T5 é selecionado um novo processador também do tipo P2 depois da atualização de TR e CR, conforme mostrado a seguir:

TR(T5, P1) = 11 – 6 = 5 $\quad$ CR(T5, P1) = $\lceil 5 / 10 \rceil$ * \$8.00 = \$8.00

TR(T5, P2) = 15 – 8 = 7 $\quad$ CR(T5, P1) = $\lceil 7 / 10 \rceil$ * \$5.00 = \$5.00

O custo final do sistema é \$23.00. A solução gerada com processadores de tipos diferentes tem custo menor que a solução formada somente com processadores de um único tipo. O projeto com processadores do tipo P1 necessita de 3 processadores com um custo de \$ 24.00 e o projeto com processador do tipo P2 utiliza 5 processadores e tem um custo de \$25.00.

A Figura 5.7 mostra o escalonamento do grafo da Figura 5.1 para a mesma latência de estágio (50) e biblioteca de processadores usados na síntese com o algoritmo MPS, na Figura 5.5. As métricas da síntese do sistema gerado pelo algoritmo MCS são mostradas a seguir:

Número de processadores: 2

Custo: \$29.00

Número de estágios: 2

Latência Total: 100

Utilização Média: 55,3 %

Pode-se notar que o número de processadores neste projeto é maior que o obtido com o MPS, mas o custo final é menor. A latência deste projeto, 100 unidades de tempo, é maior que com o

algoritmo MPS, o que pode ser explicado pelo fato de que os processadores selecionados pelo MCS serem mais lentos. Entretanto, a utilização média dos processadores foi praticamente igual, 53% para o algoritmo MPS e 55,3% para o MCS.

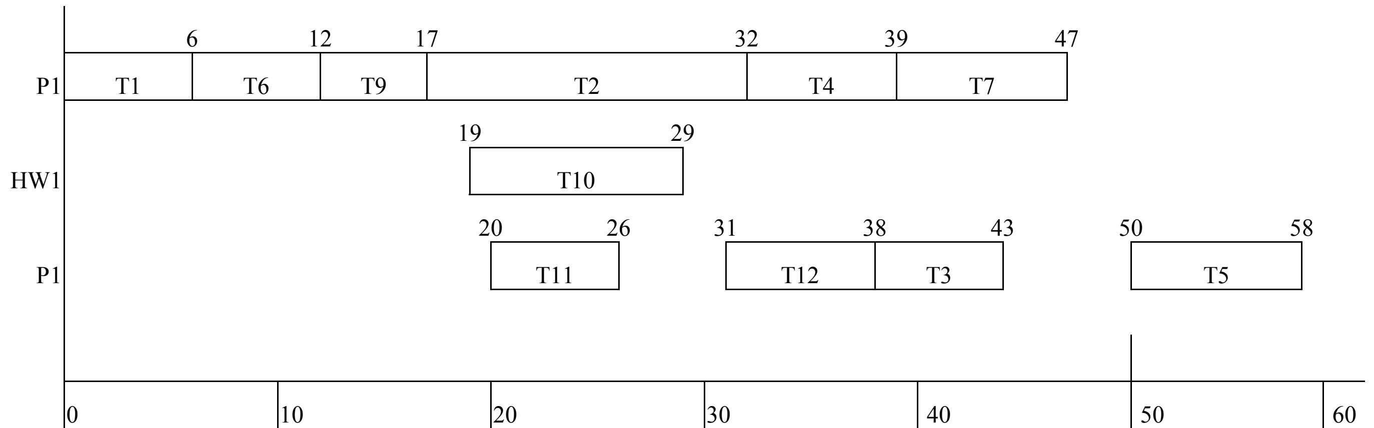


Figura 5.7 – Escalonamento do grafo na Figura 5.1, usando a Biblioteca de Componentes, aplicando o algoritmo MCS

5.3.4.3 Síntese de Pipeline com Minimização da Latência do Sistema (MLS)

A latência de um sistema com uma estrutura em *pipeline* depende do número de estágios do *pipeline* e da latência do estágio. Como a latência do estágio é determinada a partir da especificação do sistema, o algoritmo de Síntese de Pipeline com Minimização da Latência do Sistema (MLS) tem como objetivos diminuir a latência total do sistema, reduzindo o número de estágios, e minimizar o custo do sistema sempre que isto não implicar em aumento no número de estágios. A fim de se atingir estes objetivos, dois fatores que contribuem para aumentar o número de estágios devem ser minimizados: o tempo de execução das tarefas e os tempos de comunicação entre tarefas.

A heurística para atingir estes objetivos consiste de três etapas. Na primeira etapa, o algoritmo cria grupos de tarefas seguindo os caminhos de execução no grafo. A segunda etapa consiste do mapeamento de cada grupo num único processador para eliminar os tempos de comunicação entre as tarefas do grupo. Geralmente, o mapeamento de um grupo em um processador pode requerer iterações entre estas duas etapas, pois a dependência entre tarefas de grupos diferentes pode atrasar a execução de tarefas excedendo a duração do estágio. Na terceira etapa, o algoritmo reduz o número de processadores alocados reunindo, quando possível, em um único processador grupos mapeados em diferentes instâncias de um mesmo tipo de processador nas etapas anteriores.

As três etapas do algoritmo são divididas em duas fases. Na primeira fase ocorre o agrupamento das tarefas e o mapeamento nos processadores. Esta fase completa a síntese de um *pipeline* para o sistema, procurando em todos os passos minimizar o número de estágios. Este *pipeline*, entretanto, pode incluir processadores com baixa utilização. A segunda fase procura aumentar a utilização dos processadores, colocando em um único processador, grupos anteriormente mapeados em processadores do mesmo tipo. Esta fase não altera o número de estágios do *pipeline* sintetizado na primeira fase.

Fase 1: Síntese Usando Agrupamento de Tarefas

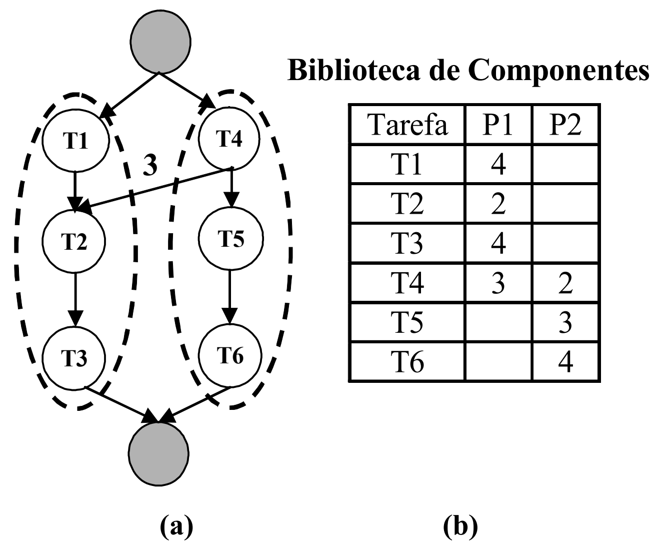
O algoritmo segue as arestas que interligam tarefas no grafo de especificação do sistema para criar os grupos. As tarefas de um grupo estão interligadas formando "uma linha de execução", isto é, de cada tarefa do grupo parte uma aresta para uma tarefa sucessora incluída no grupo. Apenas uma das sucessoras de uma tarefa é incluída no grupo da tarefa.

Após o agrupamento, os tempos de comunicação entre as tarefas dentro do grupo são considerados nulos, pois todas as tarefas do grupo serão mapeadas no mesmo processador. Entretanto, os tempos de comunicação entre as tarefas que estão em diferentes grupos não são alterados. Assim, a idéia dos grupos é definir um conjunto de tarefas a ser executado seqüencialmente num processador.

A alocação dos processadores que formarão a arquitetura final do sistema é feita durante a definição dos grupos, enquanto o escalonamento das tarefas ocorre após o agrupamento, sendo independente dele. Isto implica que, embora as tarefas sejam mapeadas nos processadores, o sistema pode não apresentar um escalonamento possível, pois as tarefas são mapeadas nos processadores assumindo que a soma de seus tempos de execução seja menor ou igual à latência do estágio, não se verificam se as dependências entre as tarefas alocadas em processadores distintos.

O exemplo da Figura 5.8 ilustra esta situação. Nele há um grafo representando um sistema com seis tarefas. A taxa de entrada dos dados é de um conjunto de dados a cada 10 unidades de tempo, o que define a latência do estágio, também, em 10 unidades de tempo. Foram criados dois agrupamentos de tarefas e os tempos de execução das tarefas T1, T2 e T3 são, respectivamente, 4, 2 e 4 unidades de tempo no processador P1. As tarefas T1 e T4 são as primeiras a serem executadas e são predecessoras da tarefa T2. Como a tarefa T4 está em um grupo diferente de T2, o tempo de comunicação entre elas deve ser considerado para calcular o instante de início de T2. Assim, a tarefa somente poderá iniciar sua execução no instante 5. Com isto, o processador P1 poderá executar, durante este estágio, somente uma tarefa com tempo de execução máximo de 3 unidades de tempo. Nesta condição não é possível escalonar a tarefa T3 neste estágio, pois seu tempo de execução é igual a 4 unidades de tempo.

Um método para permitir o escalonamento em um estágio de todas as tarefas de um grupo é reduzir a taxa de ocupação do processador associado ao grupo. A taxa de ocupação é a percentagem da latência do estágio na qual as tarefas podem ser escalonadas. Uma taxa de ocupação de 100% significa que a soma do tempo de execução das tarefas mapeadas em um processador pode ser igual à latência do estágio e uma taxa de ocupação de 50% significa que a soma dos tempos de execução poderá ser, no máximo, igual à metade da latência do estágio.



Biblioteca de Componentes

Tarefa	P1	P2
T1	4	
T2	2	
T3	4	
T4	3	2
T5		3
T6		4

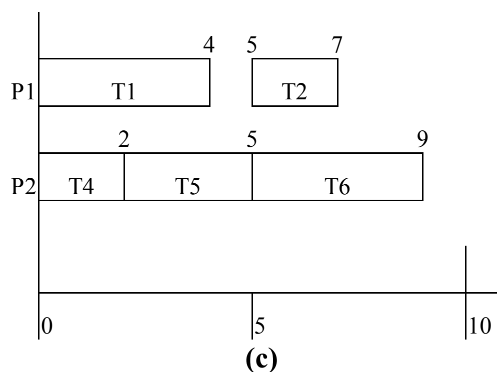


Figura 5.8 – Exemplo de síntese (c) de um grafo (a) com dependência entre tarefas de diferentes grupos. A Biblioteca de Componentes usada para a síntese é apresentada em (b).

A redução do número de tarefas no processador possibilita que haja mais intervalos de tempos disponíveis para comunicação entre tarefas aumentando as chances de se obter o escalonamento das tarefas mapeadas no processador num único estágio.

O limite para a taxa de utilização tem valor inicial igual a 100% da latência do estágio e a criação dos grupos e o escalonamento das tarefas é realizado de maneira iterativa, reduzindo o valor limite para a taxa de utilização a cada iteração até que todos os grupos consigam ser escalonados. Como em cada passo do processo a taxa de utilização máxima é reduzida, os grupos de tarefas são diferentes para cada iteração.

A definição de um grupo começa com a tarefa de maior prioridade não pertencente a algum grupo anteriormente criado. Antes de criar um novo grupo para esta tarefa, é verificado se esta pode ser incluída em algum dos grupos de suas tarefas antecessoras. Um grupo pode receber novas tarefas se a soma dos tempos de execução das tarefas do grupo não ultrapassar a taxa de utilização definida para o sistema. Um grupo que ainda aceita que novas tarefas sejam incluídas é dito um *grupo aberto* e o grupo que não aceita mais tarefas é dito um *grupo fechado*.

A busca por um grupo aberto que possa receber uma nova tarefa Ti começa ordenando suas tarefas antecessoras em ordem decrescente de tempo de comunicação destas tarefas com a tarefa Ti. É selecionada a primeira tarefa, Tj, entre as antecessoras da tarefa Ti que atenda às seguintes restrições:

a) O grupo da tarefa Tj não inclui nenhuma tarefa sucessora de Tj;
b) Existe pelo menos um processador na Biblioteca de Componentes, tal que:
    I. Existe implementação para todas as tarefas do grupo da tarefa Tj neste processador;
    II. Existe implementação para a tarefa Ti neste processador;
    III. A soma dos tempos de execução de todas as tarefas é menor que o produto da taxa de utilização do processador pela latência do estágio do *pipeline*;

A condição (a) é essencial para adicionar novas tarefas aos grupos já existentes e não permite que duas tarefas que podem executar em paralelo concorram pelo mesmo processador, pois isto poderia causar atraso na execução de uma delas. A condição (b) visa permitir o escalonamento das tarefas de um grupo no mesmo estágio do *pipeline*.

Se for encontrada alguma tarefa antecessora que atenda a estas condições, a tarefa Ti é adicionada ao grupo da antecessora no processador selecionado. Caso contrário, um novo grupo é criado para armazenar a tarefa.

Após a definição do grupo da tarefa, é modificado o estado de todos os grupos que podem ser fechados. Primeiro é testado se o grupo da tarefa recém agrupada pode ser fechado, o que ocorre se a tarefa não tiver mais nenhuma tarefa sucessora. Depois é verificado se os grupos das outras tarefas antecessoras da tarefa recém agrupada podem ser fechados. Para cada uma das antecessoras, é

verificado se todas suas tarefas sucessoras já estão agrupadas. Se esta condição for verdadeira, o grupo é fechado.

A Figura 5.9(b) apresenta um exemplo de agrupamento das tarefas de um grafo mostrado na Figura 5.9(a) com uma latência de estágio de 50 unidades de tempo usando a Biblioteca de Componentes apresentada na Tabela 5-1. Para os custos dos processadores consideramos os seguintes valores: P1 – $2.00; P2 – $5.00; P3 – $14.00 e HW1 – $25.00.

O escalonamento das tarefas segue o cálculo das distâncias mostrado na Tabela 5-2. A ordem final das tarefas na lista de tarefas, em ordem decrescente de distância, é T1, T6, T9, T2, T4, T10, T7, T11, T5, T12, T3 e T8.

O primeiro grupo é formado com a tarefa T1. As duas próximas tarefas a serem agrupadas são T6 e T9, que são chamadas de tarefa irmãs, pois possuem, ao menos, uma tarefa antecessora em comum. Por serem tarefas irmãs, são criados grupos distintos para estas tarefas para que não haja concorrência entre elas, uma vez que isto poderia implicar no atraso na execução de uma delas.

A próxima tarefa, T2, é agrupada com a tarefa T1, pois pertencem ao mesmo caminho de execução. A tarefa T4 não pode ser agrupada com a tarefa T1, pois é uma tarefa irmã da tarefa T2. Assim, um novo grupo é criado.

Para a tarefa T10 é criado um novo grupo, pois é a única tarefa a ser executada em hardware. As tarefas restantes, T7, T11 T5, T12, T3 e T8, são simplesmente adicionadas aos grupos de suas antecessoras. O escalonamento deste agrupamento inicial é mostrado na Figura 5.10. As métricas da síntese do sistema gerado são mostradas a seguir:

Numero de processadores: 4

Custo: $72.00

Número de estágios: 1

Latência Total: 50

Utilização Média: 22,8 %

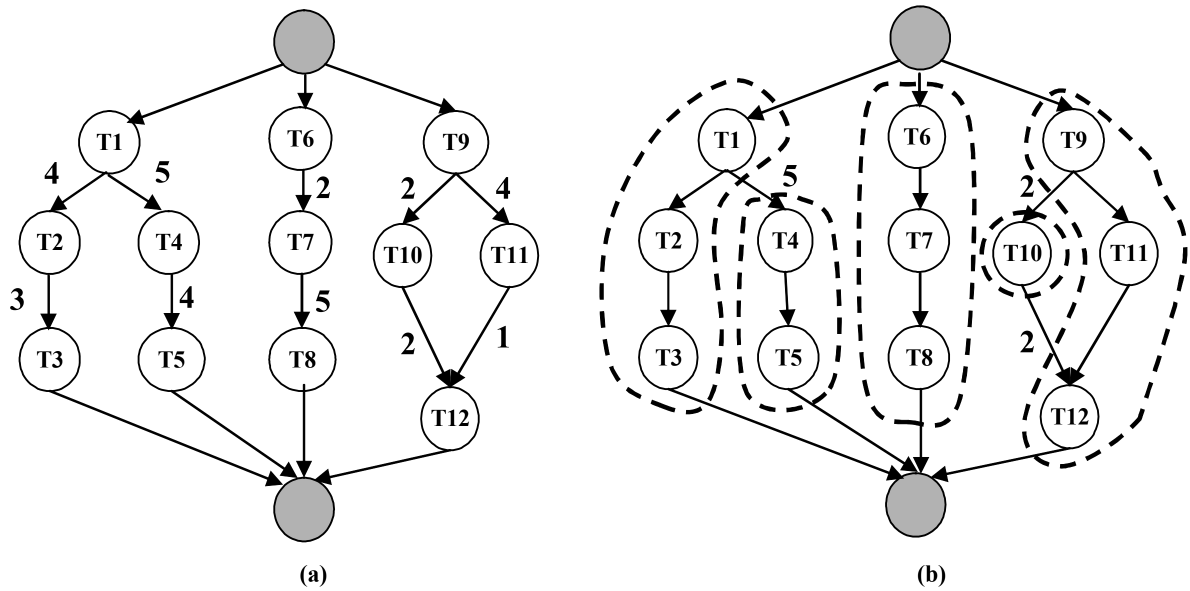


Figura 5.9 – Exemplo de um grafo (a) sem agrupamento das tarefas e (b) com agrupamento da tarefas

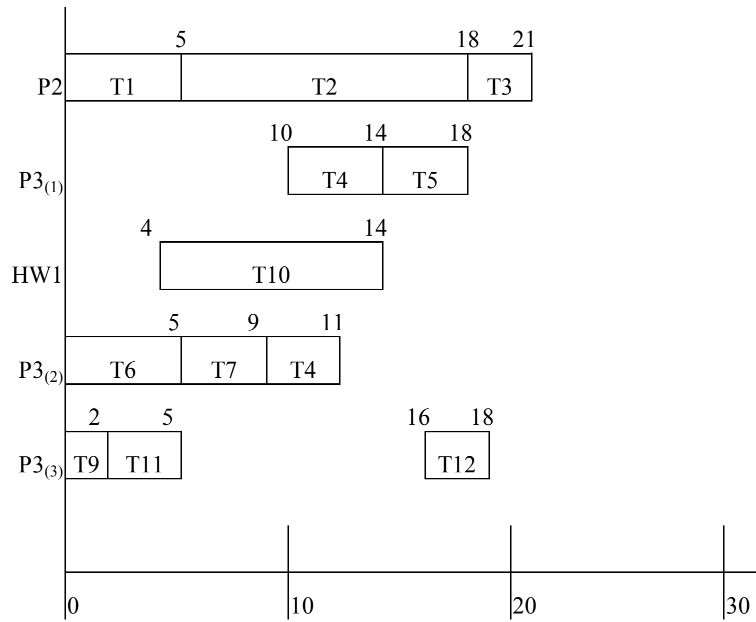


Figura 5.10 - Escalonamento para o sistema da Figura 5.9(a) após fase 1

Fase 2: Nova Síntese após Reagrupamento de Tarefas

O algoritmo descrito na fase anterior pode levar à síntese de um sistema com processadores que tem uma baixa taxa de utilização, pois podem existir grupos com pequeno número de tarefas. O propósito da próxima etapa do algoritmo é reduzir o número total de processadores do sistema mantendo-se inalterado o número de estágios, ou seja, realizar um reagrupamento das tarefas do sistema desde que a latência do sistema não seja alterada.

A redução do número de grupos é realizada unindo dois grupos para criar um grupo maior contendo as tarefas dos grupos originais. O novo grupo está associado a um único processador.

Para o reagrupamento das tarefas não existe restrição para a utilização máxima dos processadores. Uma vez que o objetivo desta etapa é reduzir o número de processadores, a utilização dos mesmos pode chegar a 100% do tempo do estágio, desde que o escalonamento seja possível e a latência total do sistema não aumente.

Para reduzir o número de processadores, o algoritmo inicialmente procura, entre os processadores alocados para o sistema, pares de processadores que sejam do mesmo tipo. A busca segue a ordem de alocação dos processadores. Para o primeiro processador, são formados pares com cada processador do mesmo tipo para o qual a soma da taxas de utilização dos processadores no par não exceda 100%. Para cada par de processadores encontrado que satisfaça esta condição, as tarefas de seus grupos são mapeadas em um único processador. É realizado então um novo escalonamento e verificado se este é melhor que o atual, aplicando-se a regra a seguir.

O agrupamento inicial é definido como sendo *o melhor agrupamento disponível*. Cada novo agrupamento formado é comparado com este agrupamento e, caso seja melhor, este o substitui. Um escalonamento E1 é considerado melhor que outro escalonamento E2 caso uma das seguintes condições ocorra:

a) O número de estágios do escalonamento E1 for menor ou igual ao número de estágios do escalonamento E2;
b) Os números de estágios dos escalonamentos E1 e E2 são iguais e o custo do escalonamento E1 for menor que o custo do escalonamento E2. O custo de um

escalonamento é igual a soma dos custos dos processadores usados para alocar suas tarefas.

Se o escalonamento for melhor, o novo agrupamento passa a ser *o melhor agrupamento disponível*. O algoritmo continua procurando pares de grupos até que não existam mais pares a serem reunidos.

Para o exemplo da Figura 5.9(a), o escalonamento após o reagrupamento das tarefas é apresentado na Figura 5.11. Comparado com o escalonamento obtido antes do reagrupamento, mostrado na Figura 5.10, tivemos uma redução de 50% no número de processadores, saindo de quatro para dois processadores.

As métricas da síntese do *pipeline* são mostradas a seguir:

Numero de processadores: 2

Custo: $44,00

Número de estágios: 1

Latência Total: 50

Utilização Média: 38 %

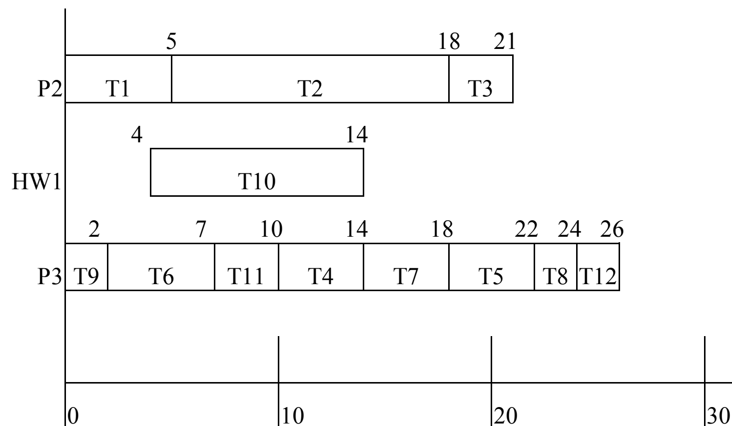


Figura 5.11 - Escalonamento para o sistema da da Figura 5.9(a) após fase 2

5.4 Conclusão

Neste capítulo foram descritos os três algoritmos desenvolvidos neste trabalho para a síntese de sistemas embutidos com organização em *pipeline* que têm por objetivo atender às restrições de desempenho dos sistemas e minimizar parâmetros específicos de projeto.

Os algoritmos compartilham uma estrutura similar em termo das entradas, das saídas e etapas da síntese. Os sistemas são representados por grafos de fluxo de dados (GFD) e as informações relacionadas às implementações das tarefas, como tempo de execução e custo, são armazenadas num banco de dados, denominado Biblioteca de Componentes.

Os algoritmos consistem de diferentes heurísticas que exploraram as características da estrutura em *pipeline* e se mostraram eficientes na alocação dos recursos e no escalonamento das tarefas nos processadores para atingir os objetivos de minimização propostos conforme mostram os exemplos de síntese apresentados no próximo capítulo.

O capítulo seguinte resume os resultados de síntese de *pipelines* realizados pelos três algoritmos apresentados neste capítulo, mostrando primeiro sistemas sintéticos e depois uma aplicação de compressão de sinais de áudio.

Capítulo 6

Resultados Experimentais

Este capítulo descreve o ambiente de desenvolvimento dos algoritmos e os testes com os exemplos de sistemas empregados para verificar o desempenho dos algoritmos e compará-los com trabalhos desenvolvidos anteriormente. Foram feitos vários testes usando sistemas descritos por grafos sintéticos de tarefas. Apresentamos também o resultado das sínteses de *pipelines* para um compressor de sinal de áudio e para um exemplo de [2], em que podemos comparar os resultados obtidos e mostrar a eficiência dos algoritmos desenvolvidos.

6.1 Ambiente de Desenvolvimento e Testes

Os programas que implementam os algoritmos foram desenvolvidos na linguagem de programação C++ e compilados com o compilador Borland C++, versão 5.5. O ambiente de desenvolvimento consiste de um microcomputador PC com processador Pentium II de 950 Mhz, 128 Mb de memória RAM com sistema operacional Windows 2000.

6.2 Sistemas Especificados por Grafos Sintéticos

6.2.1 Testes usando grafos sintéticos com menos de 20 tarefas

Nos primeiros testes, os algoritmos foram usados para o projeto de sistemas especificados por diversos grafos sintéticos com menos de 20 tarefas. Isto ajudou a corrigir pequenos erros de

programação e também comparar os resultados com outros algoritmos publicados. Para cada um dos testes, fizemos uma análise comparativa dos *pipelines* sintetizados pelos três algoritmos através das métricas de custo, número de estágios, número de processadores e taxa de ocupação média dos processadores. Os grafos que descrevem estes sistemas e os *pipelines* sintetizados pelos algoritmos são apresentados no Apêndice A.

6.2.2 Sistemas especificados por grafo com 50 tarefas

Os algoritmos foram testados usando grafos sintéticos da biblioteca de grafos, Standard Task Graph (STG), do Kasahara Laboratory [23], desenvolvidos especificamente para avaliação de algoritmos de escalonamento. Esta biblioteca dispõe de grafos gerados aleatoriamente com, no mínimo, 50 tarefas contendo de 200 a 1000 arestas. A descrição dos grafos inclui os tempos de execução das tarefas e os tempos de comunicação entre elas. Entretanto, para os testes, somente foi usada a topologia dos grafos. Os tempos de execução não foram considerados, pois os algoritmos buscam estes valores da Biblioteca de Componentes durante a síntese dos sistemas e os tempos de comunicação forma alterados para valores proporcionais aos tempos de execução das tarefas. Os resultados apresentados nesta seção são referentes a um grafo sintético da biblioteca contendo 50 tarefas e 108 arestas.

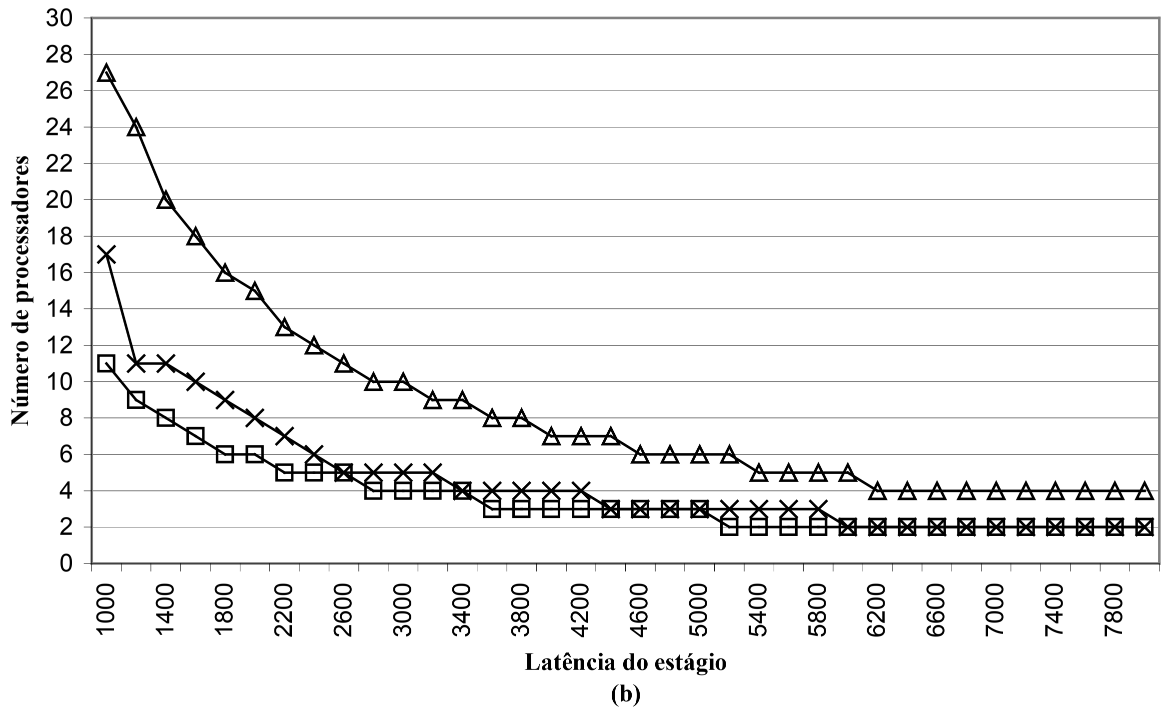
A Biblioteca de Componentes usada nos testes contém cinco processadores, designados por P1, P2, P3, P4 e P5. O processador mais rápido, P1, tem o custo mais alto. Todos os processadores apresentam implementações para as 50 tarefas do grafo. Os tempos de execução das tarefas nos processadores foram definidos da seguinte maneira. Para o processador P1, os tempos de execução das tarefas foram definidos aleatoriamente com valores entre 75 μs e 517 μs. Para cada um dos demais processadores, P2 a P5, os tempos de execução das tarefas foram aumentados com relação aos tempos de execução no processador P1. Os novos valores foram calculados multiplicando os tempos de execução em P1 por um fator aleatório que variava dentro de um intervalo definido para cada um dos processadores, conforme mostra a Tabela 6-1. Esta tabela mostra também mostra os custos dos processadores. Para garantir que os resultados apresentados possam ser reproduzidos e

comparados com outros trabalhos, os tempos de execução de cada tarefa em cada processador aparecem no Apêndice B. Aos tempos de comunicação entre as tarefas foram atribuídos valores aleatórios entre 60 μs e 150 μs. Os tempos de comunicação das arestas também são apresentados no Apêndice B.

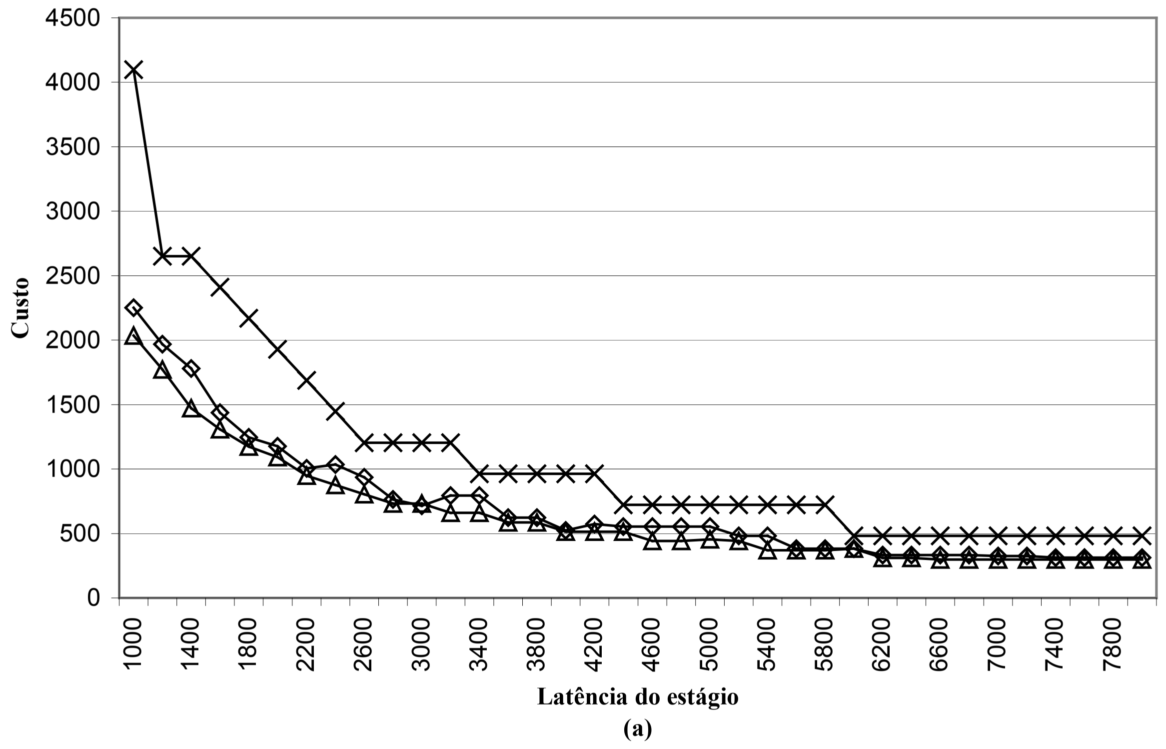
Os gráficos da Figura 6.1 resumem as métricas dos projetos de *pipeline* obtidos pelos três algoritmos de síntese para a latência de estágio do *pipeline* variando de 1000 μs a 8000 μs. São mostrados o custo do sistema, o número de processadores e o número de estágios obtidos com cada um dos algoritmos para as diferentes latências de estágio.

Tabela 6-1 - Intervalo de variação dos tempos de execução

Processador	**Intervalo de variação do tempo de execução**	
	Min.	**Max.**
P1($ 241.00)	1	1
P2 ($ 141.00)	1.1	1.8
P3 ($92.00)	1.4	2.3
P4 ($83.00)	1.8	2.5
P5 ($72.00)	2.0	3.0



(b)



(a)

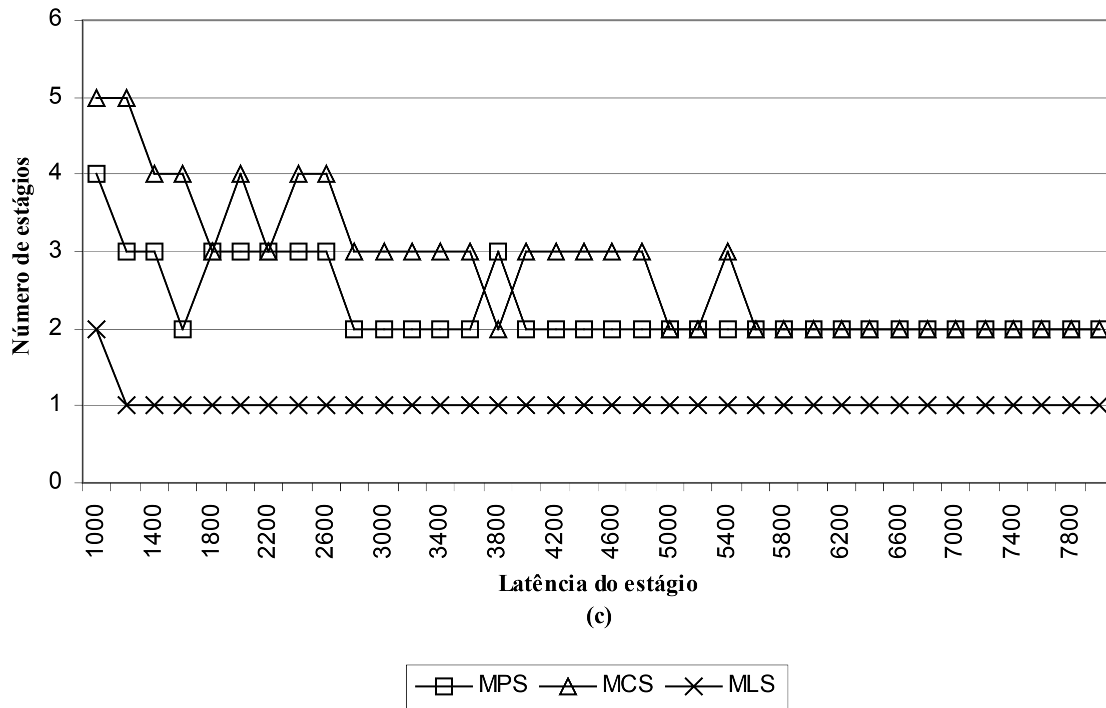


Figura 6.1 – Parâmetros de avaliação do projeto para os *pipelines* sintetizados pelos algoritmos: (a) custo, (b) número de processadores e (c) número de estágios.

Em termos de custos, os algoritmos MCS e MPS apresentaram resultados bem próximos, mas com o algoritmo MCS apresentando os melhores resultados.

Conforme mostra a Figura 6.1(a), o algoritmo de MCS obteve as sínteses com os menores custos de processadores para todos os valores de latência considerados. Os custos mais baixos são obtidos usando processadores mais lentos, o que implica em sistemas com um grande número de processadores e de um *pipeline* com muitos estágios, conforme mostram as Figuras 6.1(b) e (c).

Em relação ao número de processadores, a Figura 6.1(b) mostra que o algoritmo MPS apresentou os melhores resultados entre os 3 algoritmos. Para valores superiores a 6000 Hz, entretanto, o algoritmo MLS obteve os mesmos resultados do MPS, mas com um custo mais alto, pois, embora o número de processadores do MLS seja pequeno, os custos dos mesmos são altos.

A Figura 6.1(c) mostra que o algoritmo MLS atingiu um *pipeline* com um único estágio para valores baixos de latência dos estágios, enquanto que, para os outros algoritmos, o número de

estágios oscilou para alguns valores iniciais de latência, atingindo o valor mínimo para valores superiores a 5000 Hz. O valor mínimo conseguido, entretanto, foi superior ao atingido com o MLS.

As implementações sintetizadas pelo algoritmo MLS apresentam os menores números de estágios conseguidos com uma quantidade intermediária de processadores, mas com alto custo. Principalmente para valores baixos de latências, o custo do MLS foi superior ao dos algoritmos MCS e MPS. Por outro lado, a curva de decréscimo do custo para o algoritmo MLS apresenta decréscimos acentuados à medida que a latência do estágio aumenta.

Para os *pipelines* sintetizados pelos algoritmos MPS e MCS, o número de estágios apresenta oscilações à medida que a latência aumenta. Em outros exemplos verifica-se que o número de processadores e o custo do sistema também podem sofrer pequenas oscilações para determinados valores de latência em decorrência de determinadas características dos grafos como a dependência entre as tarefas e a ordem em que as tarefas são escalonadas. Isto nos leva a recomendar a execução dos algoritmos para valores mais baixos de latência, próximos do valor de latência especificado para comparação dos resultados.

Resta acrescentar que os algoritmos foram muito eficientes na geração dos resultados apresentando tempos de execução inferiores a 1 segundo para todos os valores de latência de estágio. Estes resultados foram obtidos mesmo usando o computador simples descrito no início do capítulo, conforme mostra a Figura 6.2.

6.3 Compressor de sinal digital de áudio (AC3)

A seguir são apresentados os resultados obtidos pelos algoritmos na síntese de *pipeline* para um compressor de áudio digital (AC3). O compressor AC3, especificado pela ATSC, pode codificar de 1 até 5.1 canais de uma representação PCM de um sinal de áudio em uma cadeia serial de bits a uma taxa variando de 32 kbps até 640 kbps. O canal de 0.1 refere-se a uma fração da banda do canal usada para sinais de baixa freqüência. A descrição detalhada do decodificador pode ser obtida nas referências [4] e [24].

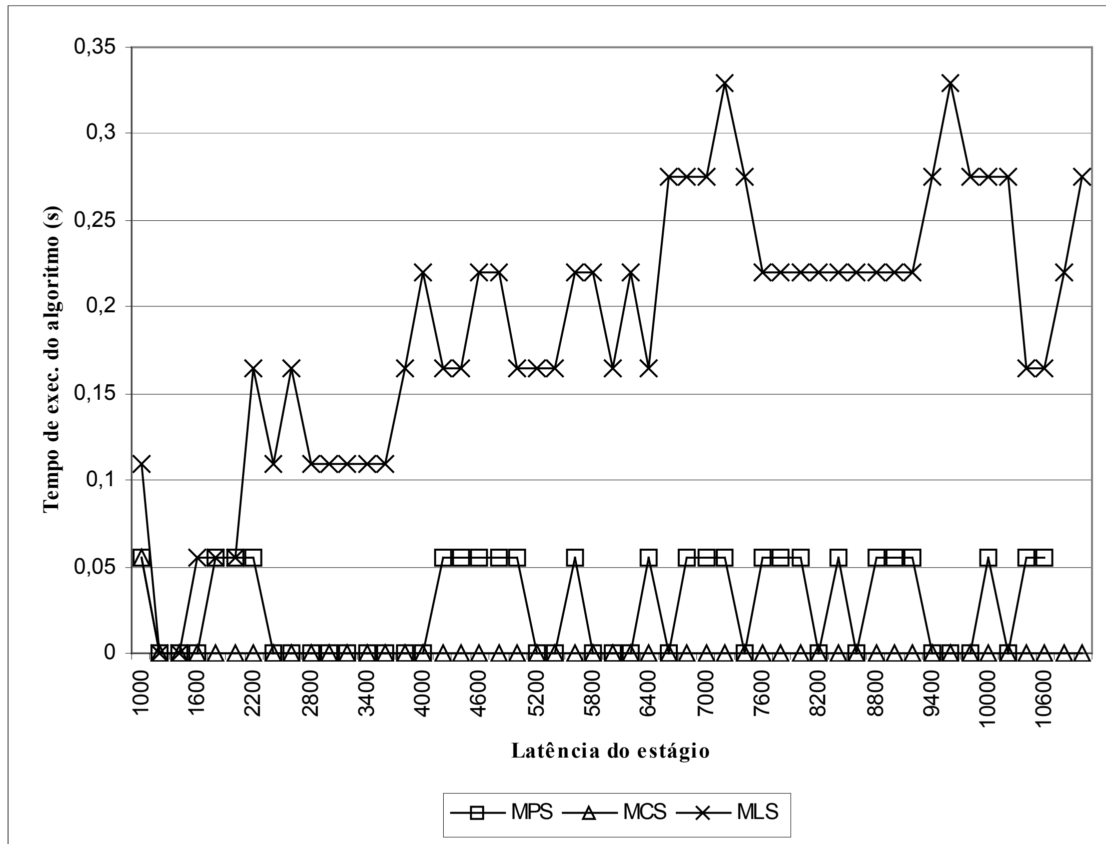


Figura 6.2 – Tempo de execução dos algoritmos para a síntese dos *pipelines*.

O grafo do compressor AC3 é composto por 26 nós, que representam diversas instâncias de seis tarefas distintas, conforme mostra a Figura 6.3. As informações transmitidas entre as tarefas correspondem aos bits usados para codificar cada amostra do sinal. Assim, a quantidade de informação transmitida entre as tarefas é igual à quantidade de bits transferidos entre as tarefas multiplicada pela quantidade de amostras processadas em uma iteração do algoritmo. A Tabela 6-2 apresenta os cálculos dos tempos de comunicação considerando as amostras e o número de bits transferidos, num canal com a taxa de 1 Mbits/s. Os valores dos tempos de comunicação estão expressos em nanossegundos.

A Biblioteca de Componentes para este teste contém 4 processadores, designados de P1 a P4, que dispõe de implementações para todas as tarefas do sistema, além de uma implementação em

hardware para a tarefa *Bit Allocation* (BA). A Tabela 6-3 apresenta os tempos de execução para as implementações das tarefas definidas na Biblioteca de Componentes. Os custos dos processadores são também mostrados na Tabela 6-3.

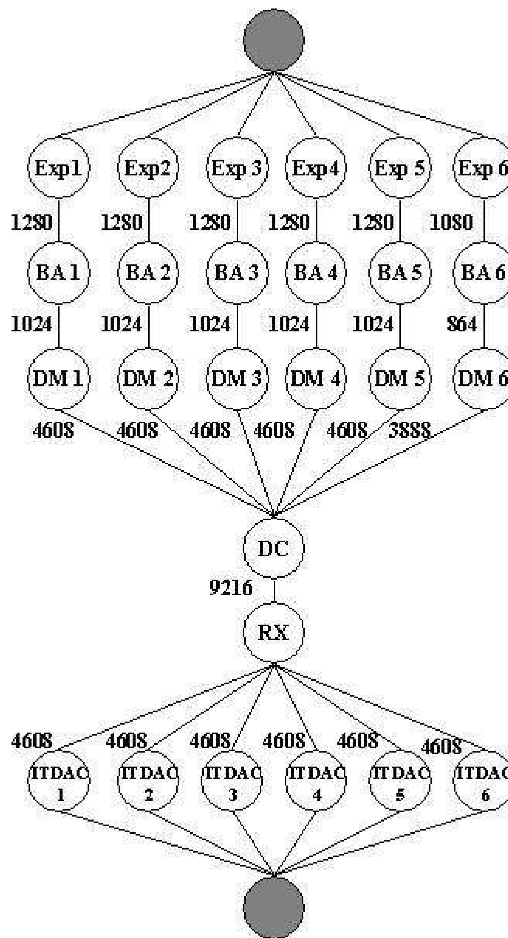


Figura 6.3 - Grafo do compressor de áudio AC3

Tabela 6-2- Cálculo dos tempos de comunicação

Bits por amostra	Numero de Amostras	Total de bits	Tempo de comunicação (ns)
4	216	864	864
	256	1024	1024
5	216	1080	1080
	256	1280	1280
7	216	1512	1512
	256	1792	1792
18	216	3888	3888
	256	4608	4608
	512	9216	9216

Tabela 6-3 - Biblioteca de Componentes

Processadores / Funções	P1 ($ 5)	P2 ($ 10)	P3 ($ 12)	P4 ($ 20)	HW1 ($ 50)
EXP (Exponent)	689	498	457	350	-
BA (Bit Allocation)	4851	4087	3740	3156	160
DM (Decode Mantissas)	2710	1519	1486	1146	-
DC (Decoupling)	458	376	309	280	-
RX (rematrixing	110	104	81	63	-
ITDAC (Inverse TDAC)	1408	-	-	842	-

Conforme a especificação do decodificador AC3 [4][24], o intervalo entre a chegada dos conjuntos de amostras dos sinais na entrada do sistema é 16 ms. Este valor foi definido como limite superior para a latência do estágio. Além desta latência, os sistemas foram sintetizados para valores menores de latência, 14,40 ms e 12,80 ms que expressam, respectivamente, 10% e 20% do valor

limite para a latência. A idéia é levar em conta que devido às possíveis oscilações nos gráficos de métricas versus latência do estágio, como as observadas no exemplo sintético da seção anterior, os projetos para valores menores de latência podem apresentar valores melhores para as métricas.

A Tabela 6-4 apresenta as métricas de projeto a saber, custo, número de processadores e número de estágios para as implementações do AC3 para as 3 latências de estágio. O menor custo para a síntese foi $20.00, obtido com o algoritmo MCS para a latência de 16 ms. O algoritmo MLS obteve uma implementação com um único estágio e 3 processadores a um custo de $60.00 para latências de 14.40 ms e 16 ms. Um resultado interessante foi a implementação de 2 estágios e 3 processadores obtida para a latência de 16 ms pelo algoritmo MPS a um custo de $45.00.

Tabela 6-4 - Métricas de projeto para os *pipelines* do compressor de áudio AC3

Latência do estágio (ms)	MPS			MCS			MLS		
	Processadores	Custo	Estágios	Processadores	Custo	Estágios	Processadores	Custo	Estágios
12.80	3	$60.00	2	5	$25.00	3	6	$120.00	1
14.40	3	$45.00	3	5	$25.00	2	3	$60.00	1
16.00	3	$45.00	2	4	$20.00	2	3	$60.00	1

6.4 Comparação com trabalhos anteriores

No teste mostrado a seguir fazemos a comparação entre a síntese obtida pelo algoritmo desenvolvido pelos autores [2] e pelos algoritmos desenvolvidos neste trabalho para um grafo simples de 5 tarefas, mostrado na Figura 6.4.

A definição das tarefas na Biblioteca de Componentes e as informações relativas aos processadores, inclusive os custos, mostrados na Tabela 6-5, também foram extraídos do exemplo em [2]. A biblioteca consta de 3 processadores com implementações para todas as tarefas do grafo, além de implementações em *hardware* para as tarefas C e E. Os tempos de execução das tarefas estão expressos em ms.

A Figura 6.5 mostra o escalonamento do grafo apresentado no artigo, na Figura 6.5 (a), e os escalonamentos obtidos com os algoritmos desenvolvidos neste trabalho, Figura 6.5 (b-d). A latência do estágio do *pipeline* é de 4000 ms.

A Tabela 6-6 compara as métricas de projeto para todas as sínteses obtidas para o exemplo. As sínteses obtidas com nossos algoritmos apresentaram resultados superiores ao algoritmo do artigo [2], tanto para o número de processadores, quanto para o custo da solução. O número de estágios praticamente não sofreu alteração, com exceção da síntese gerada pelo algoritmo MLS que obteve uma síntese com o menor número de estágios.

Tabela 6-5 - Biblioteca de Componentes

Processador / Tarefa	Pentium ($90.00)	Powerpc ($75.00)	P68000 ($60.00)	HW ($90.00)
A	3100	3800	6000	-
B	1400	2200	2800	-
C	8400	12000	18900	114
D	900	1000	1200	-
E	1230	14870	21080	224

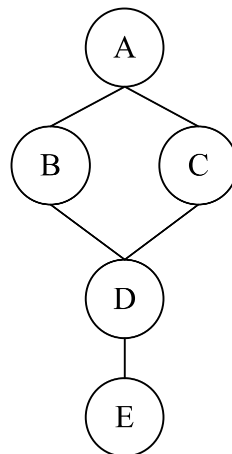
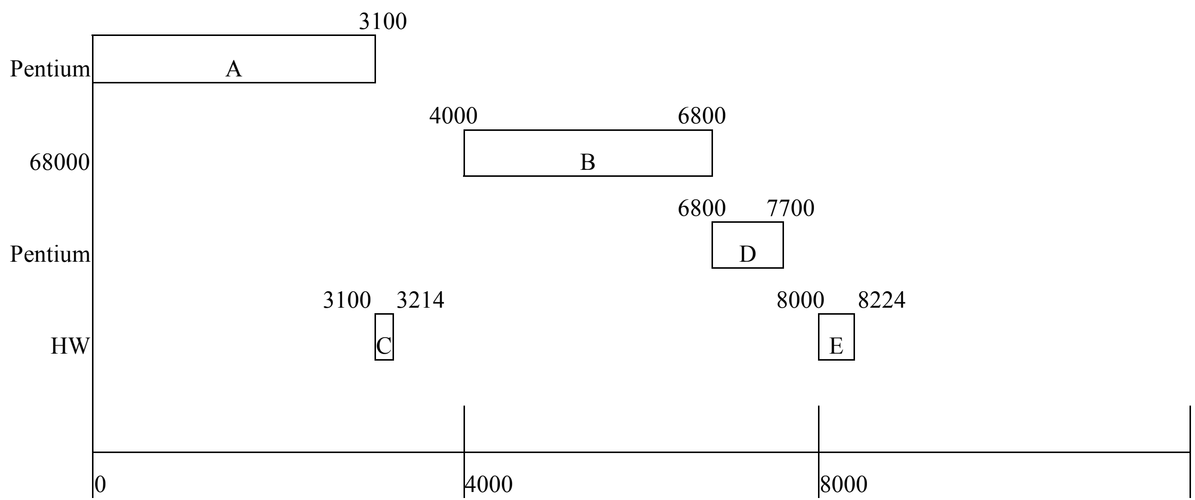
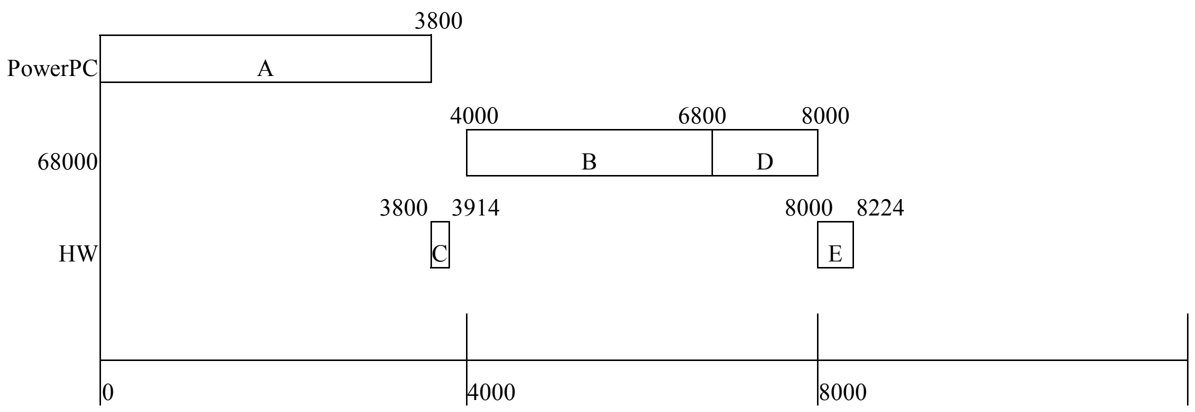


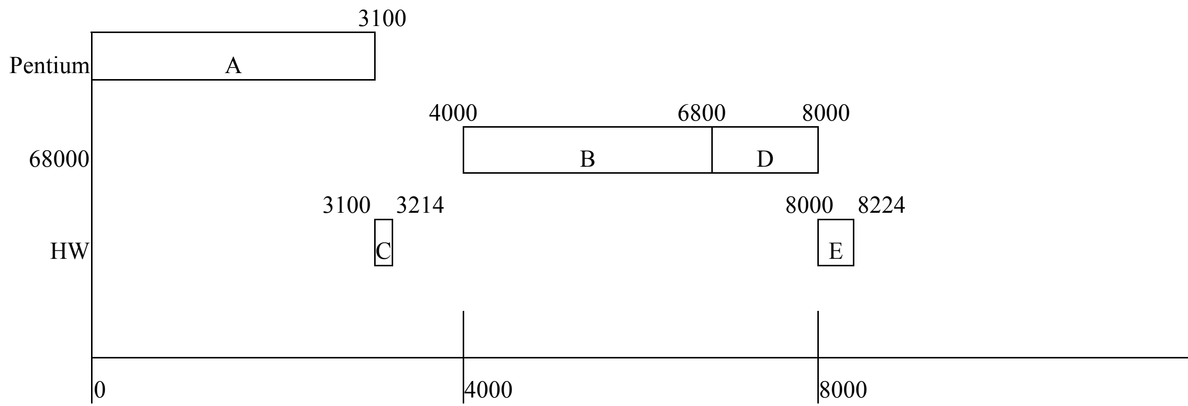
Figura 6.4 - Grafo de tarefas [2]



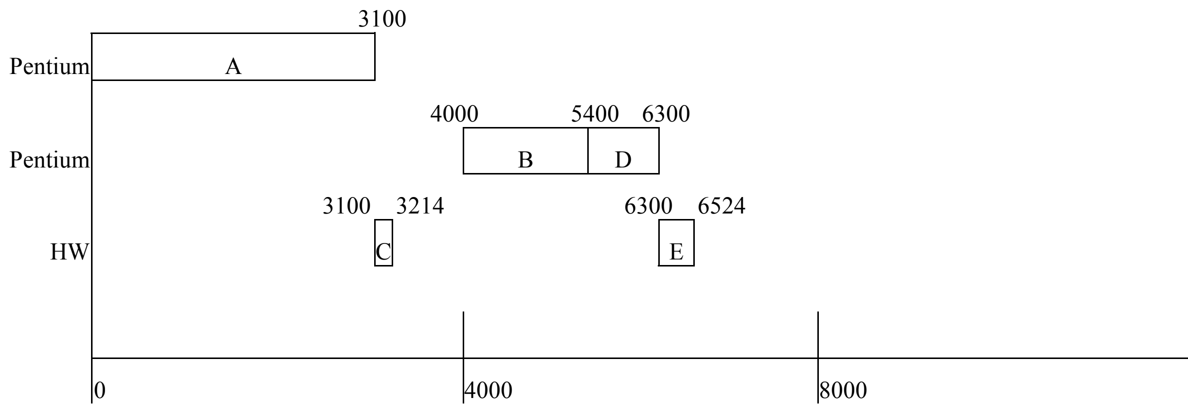
(a)



(b)



(c)



(d)

Figura 6.5 - Escalonamento obtido pelo algoritmo apresentado no artigo [2] (a) e pelos algoritmos MPS (b), MCS (c) e MLS (d).

Tabela 6-6 - Métricas de projeto

Métricas de projeto / Algoritmo	Número de processadores	Custo	Número de estágios	Taxa de utilização média
Algoritmo do artigo [2]	3	$ 330.00	3	44,61%
MPS	2	$ 225.00	3	67,81%
MCS	2	$ 240.00	3	61,98%
MLS	2	$270.00	2	47,81%

6.5 Conclusões

Nesta seção foram apresentadas as sínteses de *pipeline* realizadas usando os algoritmos descritos no capítulo anterior.

Nos testes com o sistema especificado por um grafo sintético de 50 tarefas pudemos comparar os *pipelines* sintetizados pelos 3 algoritmos. Os algoritmos atingiram os objetivos esperados na minimização das métricas específicas.

Os resultados para a síntese do compressor AC3 mostraram que os algoritmos conseguiram sintetizar um sistema real minimizando as métricas específicas para a latência de estágio máxima permitida pela aplicação. Os resultados conseguidos para os outros valores de latência permitiram considerar outras implementações para a aplicação.

A comparação com a síntese obtida em trabalhos anteriores mostrou a eficiência dos algoritmos na síntese do sistema em questão.

O próximo capítulo resume as contribuições deste trabalho e as propostas de trabalhos futuros para aprimorar os algoritmos.

Capítulo 7

Contribuições e trabalhos futuros

Neste capítulo são apresentadas as contribuições deste trabalho e as propostas de trabalhos futuros para aprimorar os algoritmos.

7.1 Contribuições

Neste trabalho foram propostas heurísticas para a síntese de sistemas embutidos com uma estrutura em *pipeline* de processadores para a execução das funções de um sistema. Em outros trabalhos, a execução em *pipeline* mostrou ser uma técnica eficiente e que possibilita gerar resultados com menor custo para a síntese de sistemas que apresentam como especificação de desempenho, o intervalo mínimo entre a recepção de conjuntos de dados de entrada [1][2][5][16].

Como resultado deste trabalho foram desenvolvidos 3 algoritmos, baseados em diferentes heurísticas para a síntese de sistemas embutidos que realizam particionamento das tarefas, alocação de processadores, mapeamento e escalonamento das tarefas com o objetivo de satisfazer as especificações de desempenho, ao mesmo tempo em que buscam a minimização de métricas específicas do projeto: custo, número de processadores ou latência do sistema.

Em todos os algoritmos, a alocação dos processadores ocorre junto com o mapeamento e o escalonamento das tarefas. O algoritmo MPS prioriza a substituição dos processadores já alocados sobre a alocação de novos processadores para minimizar o acréscimo de novos processadores. A abordagem empregada no algoritmo MCS procura minimizar o aumento do custo do sistema para alocar uma nova tarefa, aplicando o conceito de custo remanescente, que é uma estimativa do custo

dos processadores necessários para mapear as tarefas não-mapeadas do sistema, calculado sempre um novo processador tem de ser alocado. No algoritmo MLS foi empregada uma metodologia dividida em etapas, na qual é gerado um escalonamento inicial das tarefas do sistema que atende aos requisitos de tempo. Posteriormente, o escalonamento é otimizado buscando-se uma diminuição no custo da solução, através da redução do número de processadores alocados.

Os resultados apresentados pelos algoritmos nas sínteses de sistemas realizadas, tanto com as aplicações sintéticas como com a aplicação real mostraram que estes são eficazes na síntese dos sistemas e na minimização das respectivas métricas de sistema.

7.2 Trabalhos futuros

Existem várias propostas para futuros trabalhos que permitirão melhorar os resultados das sínteses obtidas e aprimorar os algoritmos para que estes possam abranger outras classes de aplicações. Algumas propostas se aplicam a todos os algoritmos, enquanto outras são específicas para um determinado algoritmo.

O algoritmo MPS pode se beneficiar de outros critérios para a escolha do novo processador quando for necessário fazer a substituição dos processadores, que não considerem somente o custo dos processadores, mas também busquem melhorar o desempenho global da síntese.

Podem ser desenvolvidas novas funções de custo para o algoritmo MCS que considerem, além do custo do processador, o custo de outros elementos como memória e os custos da infra-estrutura de interconexão entre os processadores. Para um sistema integrado num único chip (SOC), o custo pode contabilizar a área do silício. Este algoritmo poderá também ser adaptado para levar em conta o custo associado ao consumo de energia.

No algoritmo MLS, novos critérios, bem como valores distintos de limiar, podem ser aplicados na formação dos grupos e no reagrupamentos das tarefas com o objetivo de reduzir o custo da síntese e melhorar o aproveitamento dos processadores.

Entre os trabalhos futuros que se aplicam a todos os algoritmos podemos citar a elaboração de novos critérios para ordenação da fila de tarefas prontas que explorem as características implícitas da

arquitetura *pipeline*. Outra linha de investigação é a síntese de *pipelines* para sistemas especificados por múltiplos grafos com diferentes taxas de chegada de dados adotando soluções baseadas no princípio de otimalidade de Pareto.

Por último, podemos citar a possibilidade da integração dos algoritmos de síntese de *pipeline* de sistema com outras heurísticas que sintetizam *pipeline* de registradores,como as descritas em [1] [9].

Referências Bibliográficas

[1] S. Bakshi, D. Gajski, "Hardware/software Partitioning and Pipelining," *Proceedings of the IEEE/ACM Design Automation Conference* (DAC'97), IEEE, 1997.

[2] S. Bakshi, D. Gajski, "Partitioning and Pipelining for Performance Constrained Hardware/Software Systems", *IEEE Transactions on VLSI Systems*, vol. 7, no. 4, Dec. 1999.

[3] L. Bianco, M. Auguin, G. Gogniat, A. Pegatoquet, "A Path Analysis based Partitioning for Time Constrained Embedded Systems," *6th International Workshop on Hardware/Software Co-design*, 1998.

[4] L. Bianco, M. Auguin, A. Pegatoquet, "A Prototyping Method of Embedded Real Time Systems for Signal Processing Applications", *25th Euromicro Conference*, vol. 1, pp. 1318, 1999.

[5] K. Chatha, R. Vemuri, "Hardware-Software Partitioning and Pipelined Scheduling of Transformative Applications", *IEEE Transactions on VLSI Systems*, vol. 10, no. 3, June 2002.

[6] B. P. Dave, G. Lakshminarayana, N. K. Jha, "COSYN: Hardwre-Software Co-synthesis of Embedeed Systems", *Proceedings of Design Automation Conference*, pp. 703-708, 1997.

[7] A. Doboli, P. Eles, "Scheduling Under Data and Control Dependencies for Heterogeneous Architectures", *Proceedings of IEEE International Conference on Computer Design*, 1998.

[8] R. Ernst, "Codesign of Embedded Systems: Status and Trends", *IEEE Design &Test of Computers*, Vol. 15, No 2, pp. 45-54, 1998.

[9] D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, 1994.

[10] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, NY, 1979.

R. Govindarajan, E. R. Altman, G. R. Gao, "Co-Scheduling Hardware and Software Pipelines", *Proceedings of Second IEEE Symposium on High-Performance Computer Architecture*, pp 52-61, 1996.

[11] J. Grode, P. V. Knudsen, . Madsen, “Hardware Resource Allocation for Hardware/Software Partitioning in the LYCOS System”, Design *Automation and Test in Europe (DATE '98),* pp. 22, 1998

[12] J. L. Hennessy, D. A. Patterson, *Computer Organization and Design: theHardware/Software* I*nterface*, Morgan Kaufmann Publishers, 1998.

[13] A. Kavalade, E. A. Lee, “A Hardware/Software Codesign Methodology for DSP Applications”, *IEEE Design & Test of Computers*, pp 16-28, Sept. 1993.

[14] A. Kalavade, E. A. Lee, “A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem”, *Proceedings of Codes/CASHE ´94, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 22-24, pp 42-48, 1994.

[15] H. Liu, D. F. Wong, “Integrated Partitioning and Scheduling for Hardware/Software Co-design“, *Proceedings of International Conference on Computer Design*, Oct. 1998.

[16] P. Palazzari, L. Baldini, M. Coli, “Synthesis of Pipelined Systems for the Contemporaneous Execution of Periodic and Aperiodic Tasks with Hard Real-Time Constraints”, *Proceedings of 18th International Parallel and Distributed Processing Symposium(IPDPS’04)- Workshop 2*, p. 121a, 2004.

[17] S. Ranaweera, D. P. Agrawal, “Scheduling of Periodic Time Critical Applications for Pipelined Execution on Heterogeneous Systems“, *Proceedings of the 2001 International Conference on Parallel*., April 2001, pp. 131-140.

[18] B. R. Rau, C. D. Glaeser, “Some Sheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing”, *Proceedings of 14th Annual Workshop on Microprogramming*, Dec. 1984, pp. 183-198.

F. Slomka, M. Dorfel, R. Munzenberger, R. Hofmann, “Hardware/Software Codesign and Rapid Prototyping of Embedded Systems”, *IEEE Design & Test of Computers*, pp 28-38, April-June 2000.

[19] A. S. Tanenbaum, *Organização Estruturada de Computadores*, 4ª edição, Livros Técnicos e Científicos Editora, 2001.

[20] F. Vahid, T. Givargis, *Embedded Sytem Design: A unified hardware/software introduction*, John Wiley & Sons, Inc., 2002.

[21] W. H. Wolf, “Hardware-Software Co-Design of Embedded Systems”, *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967-989, July 1994.

[22] T. Yen, W. Wolf, “Performance Estimation for Real-Time Distributed Embedded Systems”, IEEE *Transactions on Parallel and Distributed Systems 9(11)*, pp 1125-1136, 1998.

[23] Standard Task Graph Set (STG), http://www.kasahara.elec.waseda.ac.jp/schedule, acessado em 01/07/2002.

[24] ATSC Digital Audio Compresion (AC-3) Standard, Revision B, http://www.atsc.org/standards, acessado em 29/09/2005.

Apêndice A

Neste apêndice são apresentados os resultados dos testes realizados para avaliar os algoritmos desenvolvidos. Os testes tinham por realizar uma comparação entre as sínteses geradas pelos algoritmos para diferentes tipos de grafos. Os grafos são constituídos por, no máximo, 20 tarefas.

A.1 Testes usando grafos sintéticos com menos de 20 tarefas

Os testes apresentados a seguir foram realizados com grafos contendo menos de 20 tarefas com o objetivo de analisar o comportamento dos algoritmos. Para cada um dos testes, fizemos uma análise comparativa das sínteses geradas pelos três algoritmos através das métricas de custo, número de estágios, número de processadores e taxa de ocupação média dos processadores.

Foram realizados testes com 2 grafos contendo 11 e 17 tarefas, extraídos de [15] que são mostrados, respectivamente, nas Figuras Figura A.1 e Figura A.3. A definição da Biblioteca de Componentes usada em ambos os testes, mostrada na Tabela A-1, é composta de 3 processadores e implementação em hardware para 5 tarefas, T1, T2, T5, T8 e T10. Os tempos de execução das tarefas e os tempos de comunicação estão expressos em ms.

A Figura A.2 apresenta os gráficos para a comparação das sínteses geradas pelos três algoritmos através das métricas de número de processadores, custo, número de estágios e taxa de utilização média dos processadores para o grafo da Figura A.1 e a Figura A.4 apresenta os resultados para o grafo da Figura A.3. Em ambos os exemplos, foram sintetizados *pipelines* para latências de estágio de 100 ms a 1000 ms com variação de 50 ms em 50 ms.

Os gráficos mostram a eficiência dos algoritmos na síntese dos *pipelines* e na minimização das métricas de projeto. Podemos observar que para alguns valores de latência, o algoritmo MPS conseguiu gerar sínteses com custos menores que os obtidos com o algoritmo MCS.

Tabela A-1 – Biblioteca de Componentes (tempos de execução em ms)

Processador / Tarefa	P1 ($2.00)	P2 ($3.00)	P3 ($4.00)	HW ($5.00)
T1	6	5	3	2
T2	15	13	11	8
T3	5	3	3	-
T4	7	5	4	-
T5	8	7	4	2
T6	6	4	5	-
T7	8	5	4	-
T8	5	4	2	1
T9	5	3	2	-
T10	7	4	3	2
T11	5	4	3	-
T12	7	4	2	-
T13	3	2	4	-
T14	9	6	5	-
T15	8	6	4	-
T16	4	-	5	-
T17	-	6	4	-

A análise dos gráficos que apresentam as taxas de ocupação dos processadores mostra uma curva caracterizada pelo decréscimo da taxa de utilização à medida que a latência aumenta, seguida por acréscimos súbitos. Podemos ver que nos pontos onde ocorrem os aumentos da taxa, há uma alteração na configuração dos processadores, observada nos gráficos de custo ou de número de processadores.

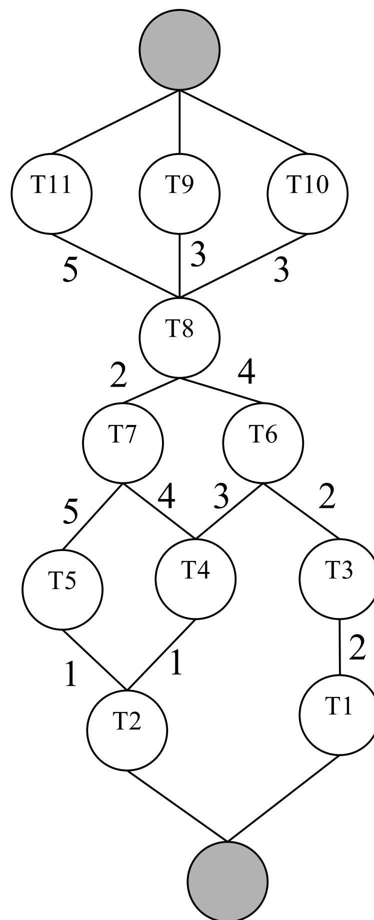
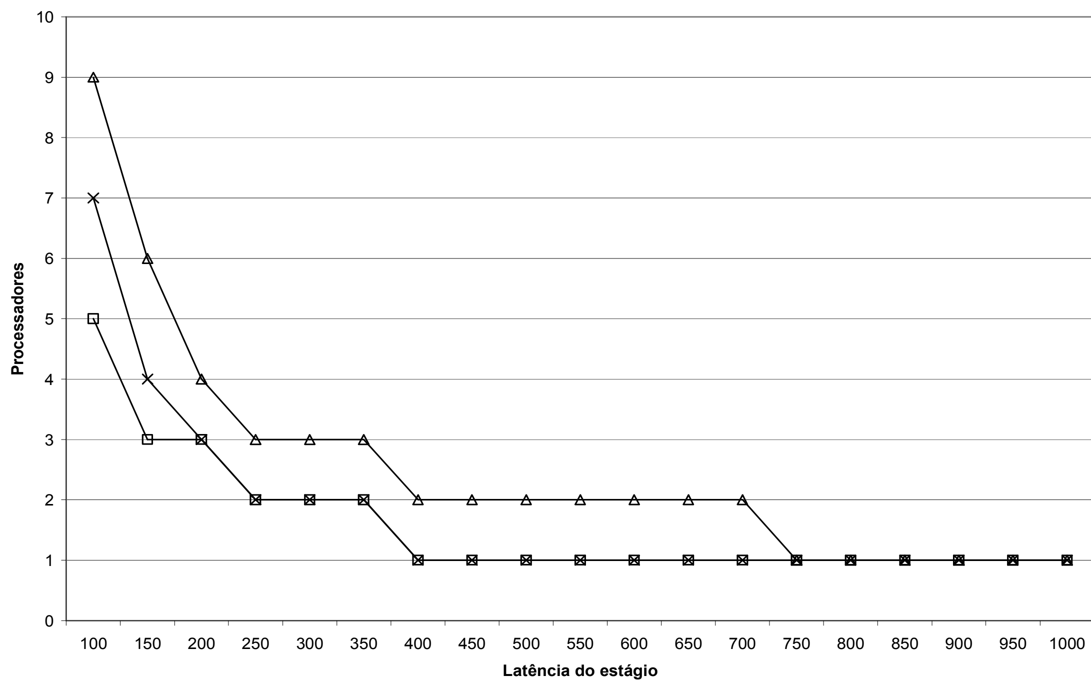


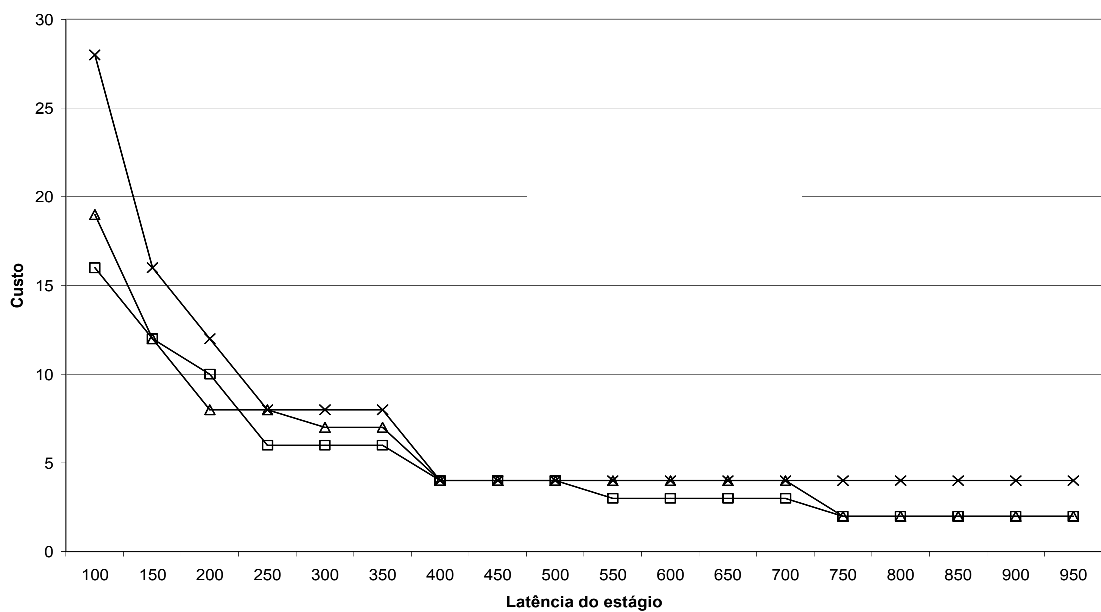
Figura A.1 – Grafo de teste com 11 tarefas [15].

A.2 Conclusão

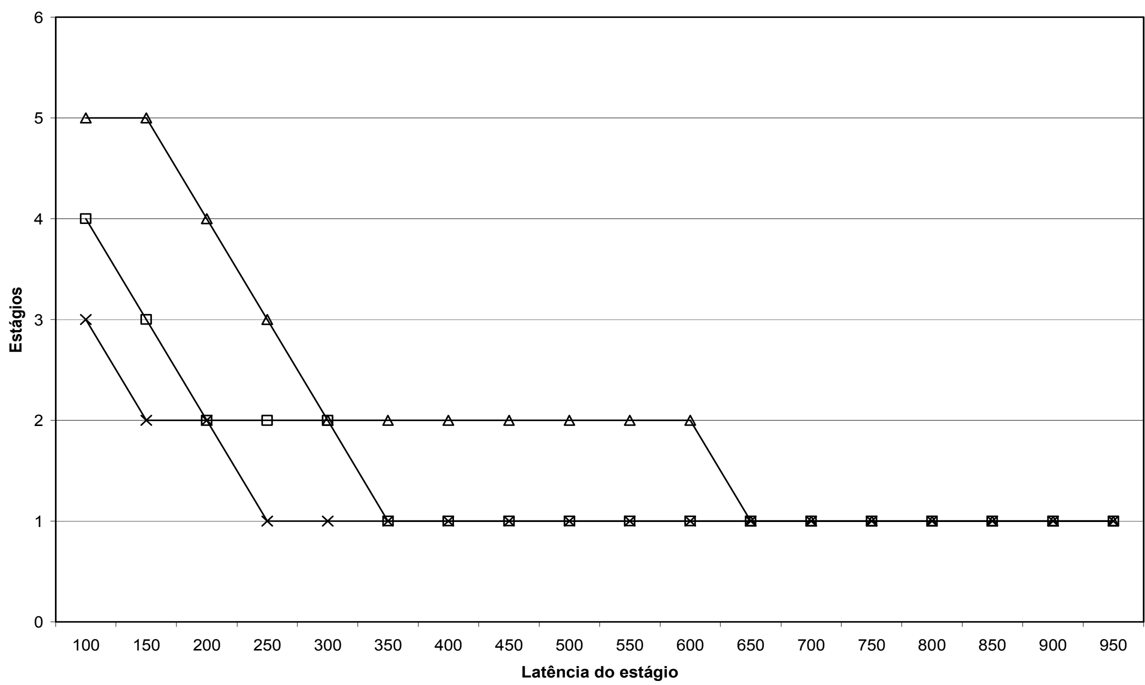
Os resultados do primeiro teste mostraram que os algoritmos desenvolvidos neste trabalho apresentaram bons resultados quando comparados com outro algoritmo de síntese de sistemas com estrutura em *pipeline* e a comparação dos resultados dos testes com os grafos sintéticos mostraram também a eficiência dos algoritmos na minimização das métricas de projeto.



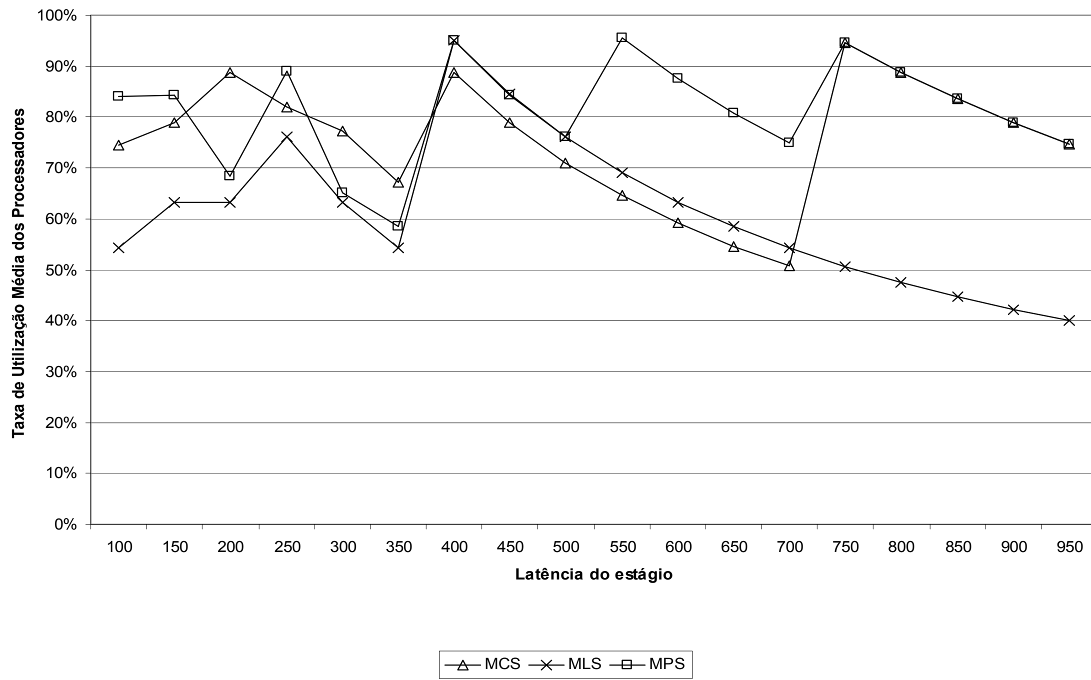
(a)



(b)



(c)



(d)

Figura A.2 – Parâmetros de avaliação do projeto para os *pipelines* sintetizados pelos algoritmos para o grafo da Figura A.1: (a) custo, (b) número de processadores, (c) número de estágios e (d) taxa de ocupação dos processadores.

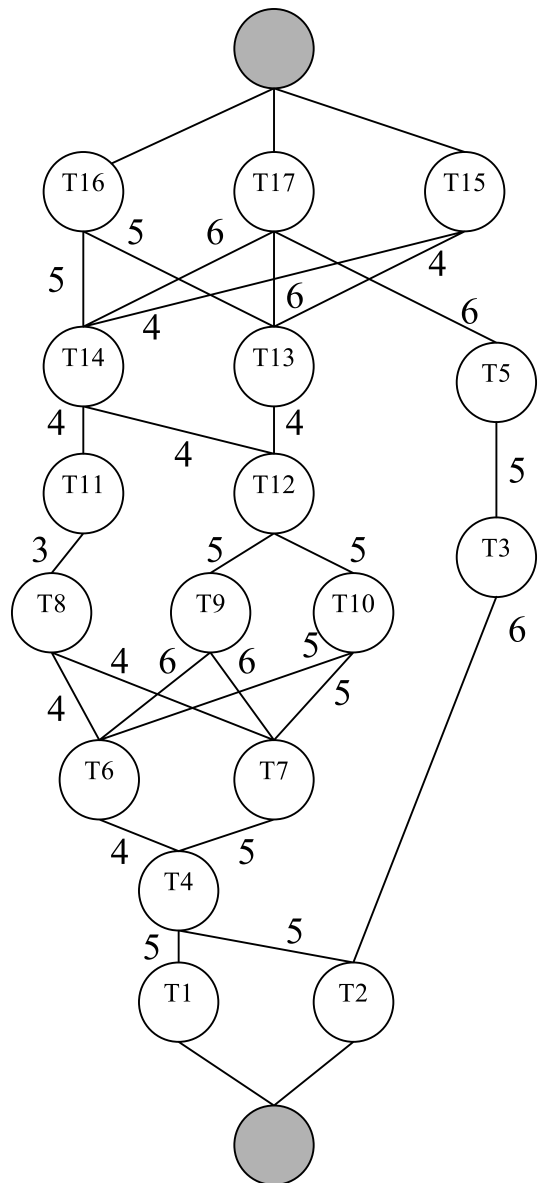
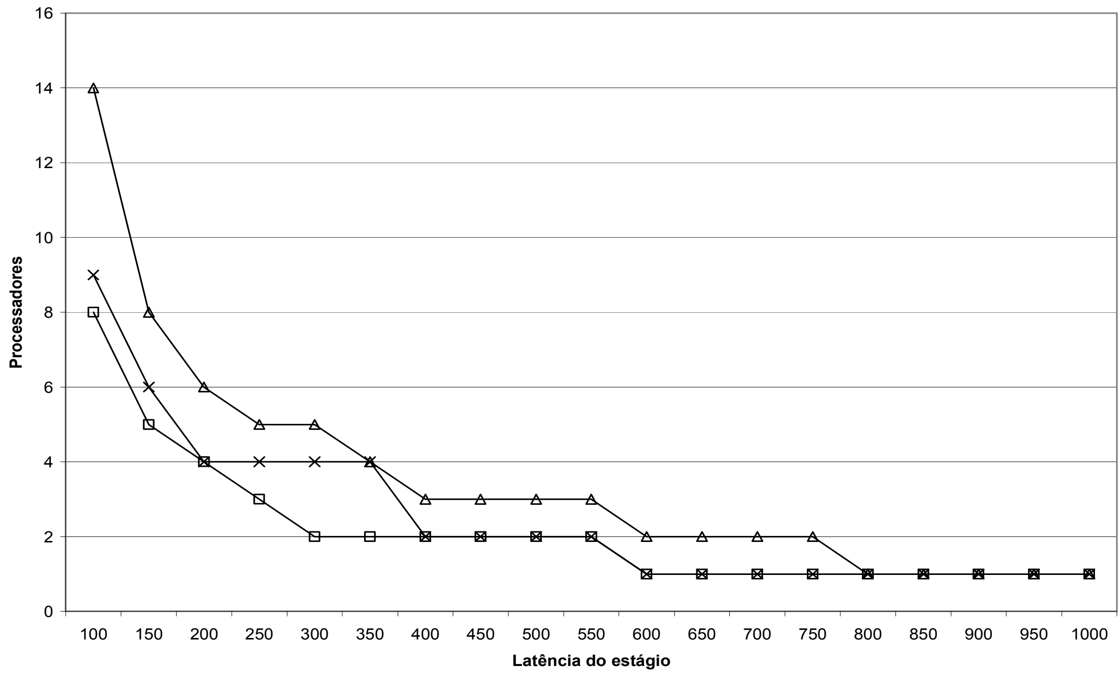
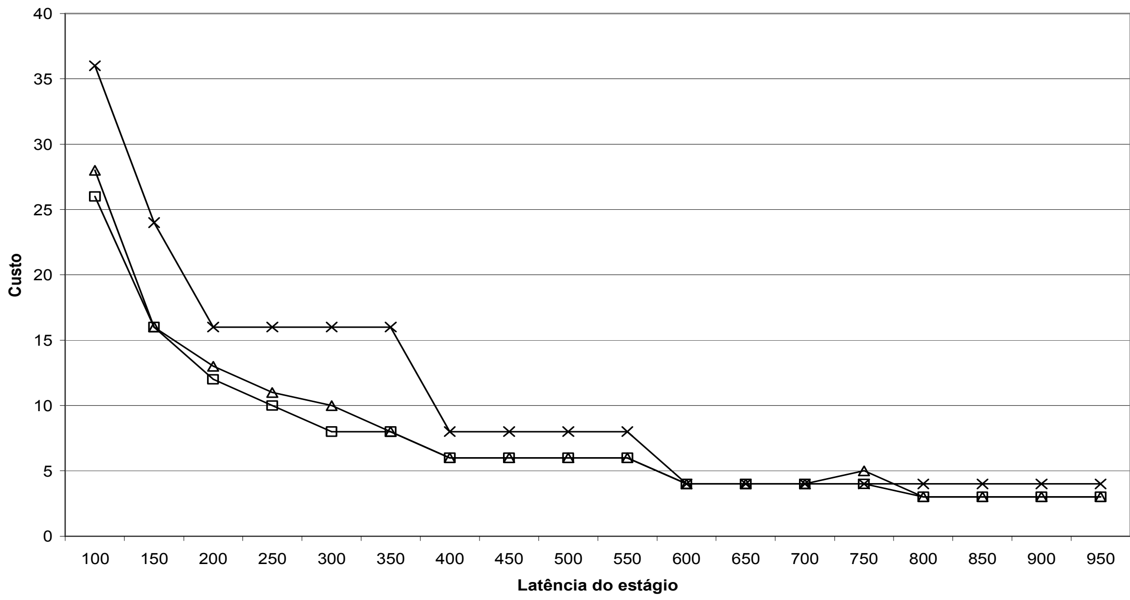


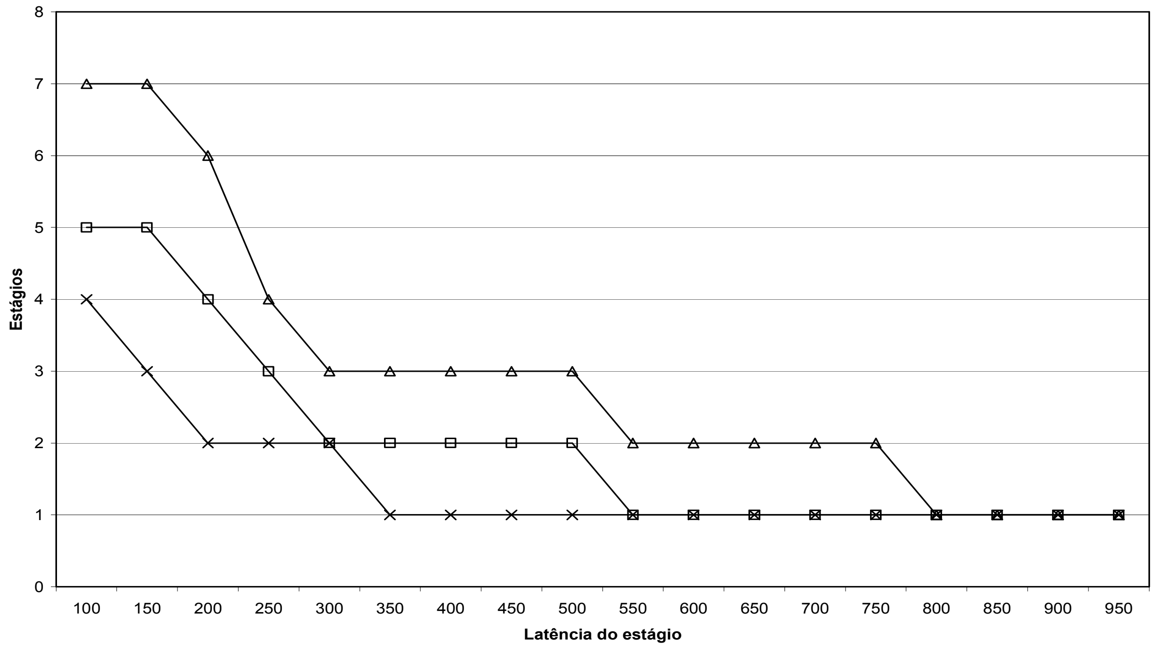
Figura A.3 – Grafo de teste com 17 tarefas [15].



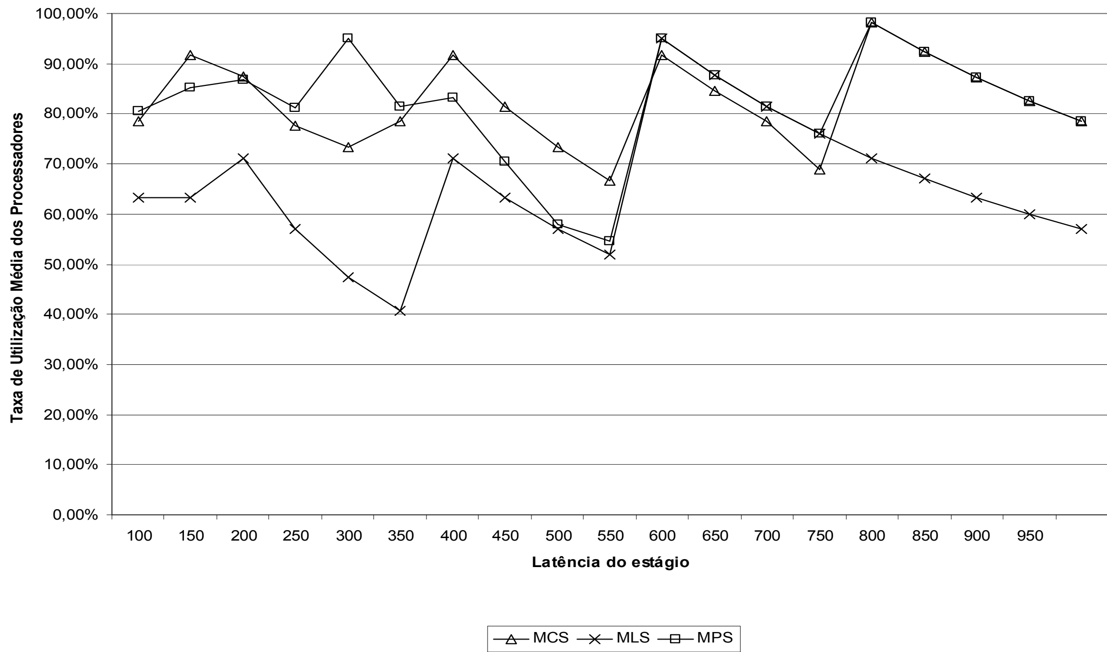
(a)



(b)



(c)



(d)

Figura A.4 – Parâmetros de avaliação dos *pipelines* sintetizados para o grafo da Figura A.3: (a) custo, (b) número de processadores, (c) número de estágios e (d) taxa de ocupação dos processadores.

Apêndice B

Neste apêndice são apresentadas as informações relativas ao grafo de tarefas e à Biblioteca de Componentes usados na seção 6.2.2, para permitir que os resultados apresentados possam ser reproduzidos e comparados com outros trabalhos.

C.1 Definição do grafo de tarefas

O grafo sintético de tarefas contém 50 nós e 108 arestas. Devido à grande quantidade de nós e arestas, será usada uma outra forma para sua representação extraída do arquivo de definição do grafo [23], que pode ser visto na Figura B.1. Cada linha do arquivo representa uma tarefa do grafo identificada pelo primeiro número na linha. O número seguinte indica a quantidade de tarefas predecessoras da tarefa. Os números seguintes indicam quais são as tarefas predecessoras e o tempo de comunicação entre elas e a tarefa. As informações são apresentadas em pares, sendo o primeiro número a tarefa predecessora e o segundo número, o tempo de comunicação entre as tarefas. Por exemplo, na sexta linha da Figura B.1, temos a tarefa 6 com apenas uma tarefa predecessora, a tarefa 1, e o tempo de comunicação entre a tarefa 6 e sua predecessora é 123.

As tarefas iniciais do grafo aparecem como tendo apenas a tarefa 0 como predecessora e com o tempo de comunicação igual a zero. As tarefas finais são colocadas como tarefas predecessoras da tarefa 51, com o tempo de comunicação também igual a zero.

```
0 0
1 1 0 0
2 1 0 0
3 1 0 0
4 1 0 0
5 1 0 0
6 1 1 123
7 1 0 0
8 1 0 0
9 1 0 0
10 1 0 0
11 1 6 102
12 1 0 0
13 3 5 95 8 50 9 130
14 2 7 118 13 120
15 1 0 0
16 1 7 123
17 2 7 90 8 112
18 1 0 0
19 1 0 0
20 2 3 61 4 89
21 4 4 131 8 114 14 136 17 92
22 2 6 131 8 65
23 4 3 123 11 113 15 103 16 87
24 2 6 124 12 113
25 3 17 123 18 141 24 66
26 3 3 95 9 130 18 90
27 3 9 139 14 79 20 132
28 2 17 131 26 150
29 1 0 0
30 2 1 66 13 91
31 3 2 109 11 124 19 60
32 3 11 148 15 150 29 55
33 1 22 126
34 4 12 148 13 84 17 148 24 65
35 1 5 72
36 2 21 87 23 141
37 3 11 140 12 68 26 120
38 2 1 73 18 118
39 5 5 105 9 99 23 70 30 107 32 85
40 3 6 140 19 136 25 149
41 4 21 135 24 68 31 84 33 103
42 1 0 0
43 2 6 59 16 120
44 2 8 89 41 96
45 4 11 122 21 51 29 72 44 142
46 4 2 69 27 101 34 75 44 105
47 5 9 140 11 110 15 94 17 90 33 97
48 3 17 112 23 91 40 134
49 3 19 139 43 70 45 96
50 2 2 140 11 82
51 13 10 0 28 0 35 0 36 0 37 0 38 0 39 0 42 0 46 0 47 0 48 0 49 0 50 0
```

Figura B.1 – Definição do grafo de tarefas

C.2 Biblioteca de Componentes

A Biblioteca de Componentes é formada por 5 processadores, denominados de P1 a P5, conforme mostrado na Tabelas Tabela B-1 e Tabela B-2. O custo dos processadores decresce de P1 até P5, sendo P1 o processador mais rápido.

Tabela B-1 – Biblioteca de Componentes

Tarefa \ Processadores	P1 ($241.00)	P2 ($141.00)	P3 ($92.00)	P4 ($83.00)	P5 ($72.00)
1	136	154	198	280	370
2	299	326	436	607	780
3	203	222	300	415	531
4	517	564	790	1056	1352
5	81	89	121	163	216
6	245	268	360	494	654
7	75	82	111	151	201
8	143	156	210	290	397
9	118	129	176	240	325
10	142	155	210	290	388
11	242	270	360	500	669
12	508	555	739	1050	1417
13	136	151	202	278	374
14	299	326	450	615	828
15	203	224	302	420	549
16	517	567	760	1059	1440
17	81	91	123	168	220
18	156	175	230	315	430
19	313	344	460	656	870
20	218	239	321	440	610
21	356	389	527	735	990
22	94	104	145	194	250
23	264	300	398	540	690
24	87	98	130	180	240
25	109	125	160	215	295

Tabela B-2 – Biblioteca de Componentes (cont.)

Processadores / Tarefa	P1 ($241.00)	P2 ($141.00)	P3 ($92.00)	P4 ($83.00)	P5 ($72.00)
26	125	138	190	245	340
27	157	172	233	310	435
28	249	273	370	480	685
29	274	317	402	540	750
30	145	159	213	280	390
31	294	322	430	578	800
32	209	230	305	410	570
33	503	551	732	1009	1390
34	90	101	135	185	240
35	181	199	270	372	490
36	245	269	360	500	678
37	75	86	112	155	205
38	143	156	210	290	390
39	106	118	160	220	280
40	278	311	405	550	770
41	102	118	160	210	280
42	124	143	181	250	340
43	118	130	180	240	320
44	76	87	115	158	210
45	149	163	220	310	400
46	125	139	198	255	340
47	145	164	215	300	400
48	239	267	356	490	630
49	240	268	355	490	635
50	139	156	215	279	380

Apêndice C

Neste apêndice são apresentadas as descrições dos algoritmos desenvolvidos neste trabalho na forma de pseudo-linguagem. Além dos algoritmos, também é descrita a rotina de escalonamento das tarefas.

C.1 Algoritmo – Minimização do Custo do Sistema (MCS)

Calcula distância das tarefas

Monta Lista de Tarefas Prontas, ordenada por distância

Calcula o Tempo Remanescente e o Custo Remanescente dos processadores

Enquanto (Lista de Tarefas Prontas < > {})

T = tarefa com maior distância da Lista de Tarefas Prontas

RP_T = relação dos processadores alocados para os quais existe uma implementação para a tarefa T

Se (RP_T < > { })

Mapeia a tarefa no processador onde a tarefa termina a execução mais cedo

Atualiza o Tempo Remanescente dos processadores

Senão

Aloca um processador com o menor Custo Remanescente

Mapeia a tarefa no processador

Atualiza o Tempo Remanescente e o Custo Remanescente dos processadores

Fim_senão

Atualiza Lista de Tarefas Prontas

Fim enquanto

C.2 Algoritmo – Minimização do Número de Processadores do Sistema (MPS)

Calcula distância das tarefas

Monta Lista de Tarefas Prontas, ordenada por distância

Enquanto (Lista de Tarefas Prontas < > { })

T = tarefa com maior distância da Lista de Tarefas Prontas

RP_T = conjunto dos processadores alocados que têm implementação para a tarefa T

Se (RP_T < > { })

Mapeia a tarefa T no processador de RP_T onde a tarefa termina a execução mais cedo, desde que respeitada a duração do estágio.

Senão

RC = lista dos processadores da Biblioteca de Componentes, em ordem crescente de custo, com implementação para a tarefa T

RP = conjunto dos processadores alocados

(1) Para (cada processador P em RP)

Busca em RC, abrangendo os processadores com custo maior que o custo do processador P, algum processador que implemente todas as tarefas de P

(2) Se (encontrou um processador)

Tenta mapear as tarefas de P + a tarefa T neste processador

(3) Se (escalonamento foi realizado)

Acrescenta o novo porcessador em RP

Mapeia as tarefas de P + a tarefa T no processador

Retira o processador P de RP

Pára a busca

Fim_se (3)

Fim_se (2)

Fim_para (1)

(4) Se (nenhum processador foi encontrado)

Aloca um processador do tipo mais barato com implementação para a tarefa T

Mapeia a tarefa T no processador

Acrescenta o processador ao conjunto RP

Fim_se (4)

Escalona as tarefas

Atualiza Lista de Tarefas Prontas

Fim_enquanto

C.3 Algoritmo – Minimização da Latência do Sistema (MLS)

Define taxa máxima de utilização dos processadores = 100%

(1)Repita

Gera agrupamento das tarefas (taxa de utilização)

Tenta **Escalonamento de tarefas**

Se (escalonamento não for realizável)

Reduz taxa de utilização dos processadores

(1)Até que (escalonamento seja realizado)

(2)Repita

RP = lista dos processadores alocados ordenados em ordem crescente de utilização

P1 = primeiro processador de RP

(3) Repita

P2 = próximo processador em RP do mesmo tipo de P1 tal que a soma (taxa de utilização de P1)+ (taxa de utilização de P2) seja menor que 100%

Se (encontrou processador)

Copia as tarefas de P2 em P1

Tenta escalonar as tarefas

Se (escalonamento for realizável AND número de estágio não aumentou)

Exclui P2 de RP

Senão

Retira de P1 as tarefas de P2

Fim_se

(3) Até que (algum reagrupamento foi feito OR todos os processadores de RP foram testados)

P1 = próximo processador de RP

(2) Até que (todos os processadores em RP tenham sido testados)

C.4 Algoritmo MLS - Gera agrupamento das tarefas (taxa de utilização)

Calcula distância das tarefas

Monta Lista de Tarefas Prontas, ordenada por distância

Enquanto (Lista de Tarefas Prontas < > { })

T = tarefa com maior distância da Lista de Tarefas Prontas

Se (tarefa não tem tarefas antecessoras)

Gera um novo grupo GT para a tarefa T

Associa ao grupo GT um processador do tipo mais barato com implementação para a tarefa T

Acrescenta o novo processador na relação de processadores associados a grupos

Senão

RA = lista das tarefas antecessoras de T, em ordem crescente de tempo de comunicação da tarefa antecessora com a tarefa T

Enquanto (tarefa T não foi mapeada AND existe uma tarefa em RA a ser testada)

TA = próxima tarefa de RA cujo grupo não inclui uma tarefa irmã da tarefa T

Se (encontrou tarefa)

Verifica se a tarefa T pode ser incluída no mesmo grupo que TA sem exceder a taxa de utilização

Se (positivo)

Inclui a tarefa no grupo

Senão

Busca um processador na Biblioteca de Componentes com implementação para as tarefas do grupo + a tarefa T considerando a taxa de utilização

Se (encontrou um processador)

Inclui T no grupo de TA

Associa o novo porcessador ao grupo

Inclui novo processadore na na relação de processadores associados a grupos

Retira o processador anterior do grupo da relação de processadores associados a grupos

Fim_se

Fim_se

Fim_enquanto

Se (tarefa não foi incorporada a nenhum dos grupos)

Gera um novo grupo GT para a tarefa T

Associa ao grupo GT um processador do tipo mais barato com implementação para a tarefa T

Acrescenta o novo processador na relação de processadores associados a grupos

Fim_se

Fim_enquanto

C.5 Escalonamento das Tarefas

```
Monta fila de tarefas prontas, ordenada por distância
Enquanto (Lista_de_tarefas_prontas <> {})
  T = Tarefa com maior distância da Lista de tarefas prontas
  Seleciona o processador associado à tarefa T
  Se (Tarefa T pode ser alocada no processador)
    Escalona a tarefa T no processador
    Atualiza Lista de tarefas prontas
  Senão
    Retorna (Escalonamento impossível)
  Fim enquanto
  Retorna (Escalonamento realizado com sucesso)
```