Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Algoritmos para Síntese de Sistemas Embutidos Tolerantes a Falhas Empregando Reconfiguração Dinâmica de FPGAs

Christian Farias da Silva

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação da Unicamp, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Engenharia de Computação.

Comissão Examinadora:

- Profa. Dra. Alice Maria B. H. Tokarnia (DCA/FEEC) - Presidente
- Prof. Dr. Mário Lúcio Côrtes (IC)
- Prof. Dr. Carlos Alberto dos Reis Filho (DSIF/FEEC)
- Prof. Dr. José Raimundo de Oliveira (DCA/FEEC)
- Prof. Dr. Marco Aurélio A. Henriques (DCA/FEEC)

Campinas, SP – Brasil
Junho de 2003

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

Si38a

Silva, Christian Farias da
Algoritmos para síntese de sistemas embutidos tolerantes a falhas empregando reconfiguração dinâmica de FPGAs / Christian Farias da Silva.--Campinas, SP: [s.n.], 2003.

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

1. Sistemas embutidos de computador. 2. Tolerância a falha (Computação). I. Tokarnia, Alice Maria Bastos H. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Os FPGAs ultrapassaram a barreira da utilização exclusiva na prototipagem de sistemas, principalmente pela flexibilidade e menor tempo de introdução ao mercado que possibilitam aos projetos. A capacidade de reconfiguração dinâmica de alguns desses dispositivos tem sido explorada para diminuir a área de hardware de sistemas embutidos.

A presença crescente de sistemas embutidos desempenhando funções críticas justifica o interesse na aplicação de técnicas de tolerância a falhas durante seus desenvolvimentos. No entanto, a necessidade de redundância, seja de recursos, informações ou tempo, é conflitante com as restrições encontradas no projeto desses sistemas, sobretudo no aspecto dimensão.

Nesta dissertação é proposto o emprego de FPGAs dinamicamente reconfiguráveis com o objetivo de reduzir a dimensão de sistemas embutidos tolerantes a falhas, utilizando-os como recursos redundantes, compartilhados, para a execução de funções do sistema em hardware dedicado. Um algoritmo de síntese no nível de sistema baseado nessa proposta é introduzido, o qual, a partir de uma especificação de sistema embutido e uma biblioteca de FPGAs, fornece ao projetista: 1) uma alocação de FPGAs e o mapeamento das funções do sistema nesses dispositivos; 2) escalonamentos para as funções do sistema e reconfigurações dos FPGAs, na ausência e presença de falhas em funções com requisito de tolerância a falhas; 3) a degradação no desempenho do sistema provocada por falhas em cada uma dessas funções.

A síntese realizada pelo algoritmo visa a obtenção de uma alocação de FPGAs de menor custo e/ou dimensão para o projeto. Resultados obtidos para um grupo de sistemas embutidos sintéticos, incluindo um exemplo baseado num sistema de processamento digital de sinais, são apresentados, comentados e comparados com os obtidos através de estratégias convencionais de redundância.

Palavras-chave: Síntese de Sistemas Embutidos, Tolerância a Falhas, FPGAs Dinamicamente Reconfiguráveis.

Abstract

The FPGAs are surpassing the exclusive use in system prototyping, mostly by the flexibility and less time-to-market they provide. The dynamic reconfiguration capability of some of them has been exploited to reduce the hardware area of embedded systems.

Embedded systems now perform critical tasks, which justifies the interest in using fault tolerance design techniques to design them. However, the necessity of resource, information, or time redundancy conflicts with stringent embedded systems design constraints, especially those related to system size.

This dissertation proposes the use of dynamic reconfigurable FPGAs as shared spares for system functions implemented in dedicated hardware, as a way to reduce the size of fault-tolerant embedded systems. A system-level synthesis algorithm supporting this proposal is introduced. Starting from a embedded system specification and an FPGA library, it produces: 1) an FPGA allocation and the assignment of system functions on it; 2) scheduling schemes for the system functions and FPGAs reconfigurations, in the presence or not of faults; 3) the system performance degradation caused by faults in its functions.

The synthesis algorithm targets an FPGA allocation with minimum cost and/or size. Results obtained for a set of synthetic embedded systems examples, including one based on a real digital signal processing system, are presented, discussed and compared to results obtained employing a conventional spare approach.

Keywords: Embedded System Synthesis, Fault-Tolerance, Dinamically Reconfigurable FPGAs.

Aos que fizeram minha pessoa,

Edna e Humberto.

Agradecimentos

A Deus, por inúmeras razões.

À minha família, em especial aos meus pais e minha avó, que não permitiram que nada me faltasse durante esses anos, da matéria ou do espírito.

À professora Alice, por me aceitar como aluno e orientar pacientemente todas as etapas deste trabalho.

À Ana Luísa e Juliana, que pavimentaram meu caminho até Campinas.

Aos amigos de ontem e hoje da República: Daniel, Júnior, Érico, Júlio 1, Eduardo, Adeilton, Escobar, Lindomberg e Júlio 2, pela convivência sempre pacífica e agradável.

Aos amigos e amigas do IMMEC, por sempre me receberem muito bem em suas reuniões e confraternizações.

Aos colegas do LCA, pela pronta ajuda sempre que necessária.

Ao CNPq, pela bolsa de estudo concedida.

A todos que, embora não citados acima, têm seus nomes escritos nas páginas que narram, em algum lugar, meus dias ao longo desses últimos anos.

"Idéias audazes são como peças de xadrez movidas para frente; podem ser batidas, mas podem iniciar um jogo vitorioso"

Johann W. Goethe
(1749-1832)

Sumário

Índice de Figuras

Lista de Abreviaturas

ALAP	*As Late As Possible*
ASAP	*As Soon As Possible*
ASIC	*Application Specific Integrated Circuit*
CPLD	*Complex Programmable Logic Device*
DAG	*Direct Acyclic Graph*
E3S	*Embedded Systems Synthesis benchmarks Suite*
EEMBC	*Embedded Microprocessor Benchmark Consortium*
EEPROM	*Electrically Erasable PROM*
EPROM	*Erasable PROM*
EST	*Earliest Start Time*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
HDL	*Hardware Description Language*
ILP	*Integer Linear Programming*
IP	*Intellectual Property*
JPEG	*Joint Photographic Experts Group*
LE	*Logic Element*
MPGA	*Mask Programmable Gate Array*
NMR	*N-Modular Redundancy*
PAL	*Programmable Array of Logic*
PDA	*Personal Digital Assistent*
PDF	Ponto de Detecção de Falha
PE	*Processing Element*
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PROM	*Programmable Read-Only Memory*
RTL	*Register-Transfer Level*
SOC	*System-On-Chip*
SPLD	*Simple Programmable Logic Device*
SRAM	*Static Random Access Memory*
TGFF	*Task Graph For Free*
TMR	*Triple Modular Redundancy*
UT	Unidade de Tempo
VHDL	*Very high-speed integrated circuit Hardware Description Language*
VLSI	*Very Large Scale Integration*

Capítulo 1

Introdução

Sistemas computacionais embutidos estão por toda parte. Do rádio-relógio que nos desperta pela manhã, ao último contato com o controle remoto da TV, à noite, são inúmeras as interações que realizamos ao longo do dia com os mais variados tipos de sistemas embutidos. Durante a década passada, tais sistemas beneficiaram-se (e contribuíram) com o aumento significativo no número de trabalhos em co-projeto hardware/software, passando a serem o alvo de muitas pesquisas dessa área. Avanços na tecnologia VLSI (*Very Large Scale Integration*) reduziram as dimensões dos sistemas embutidos, provocando a expansão de suas áreas de aplicação. A crescente demanda computacional de sistemas automotivos e de telecomunicações fez surgirem os chamados sistemas embutidos distribuídos, compostos por diversos elementos de processamento, heterogêneos, conectados através de *links* de comunicação.

A diversidade de componentes atualmente disponíveis, tanto em hardware quanto em software, torna os espaços de projeto dos sistemas embutidos bastante amplos, impossibilitando ao projetista explorar adequadamente as alternativas sem o auxílio de ferramentas computacionais. A maior capacidade de processamento de seus componentes torna-os também mais complexos. Esses fatores têm impulsionado o desenvolvimento de metodologias e ferramentas computacionais para sistemas embutidos no nível de sistema. A quantidade de elementos que devem ser manipulados diretamente pelo projetista nesse nível é significativamente menor do que em outros níveis de projeto. A Figura 1.1 ilustra esse fato, evidenciando que quanto maior o nível de abstração empregado, menor a precisão dos dados obtidos quanto à implementação do sistema, em decorrência do emprego de estimadores mais simples e rápidos.

Entre os componentes disponíveis para sistemas embutidos encontram-se os FPGAs (*Field Programmable Gate Array*), uma denominação genérica utilizada para referir-se a hardware programável ou configurável. Esses dispositivos evoluíram bastante desde o seu apare-

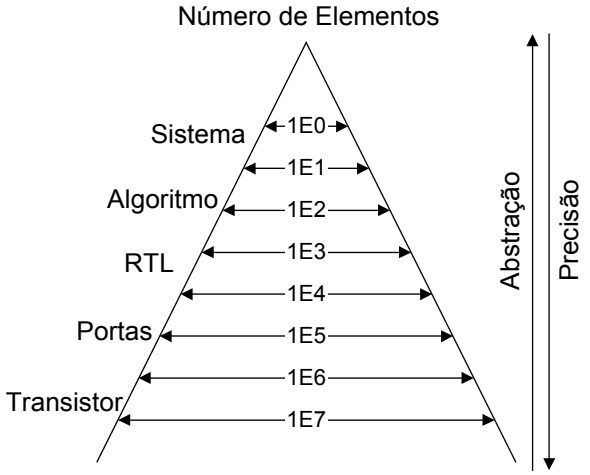


Figura 1.1 Abstração x Número de Componentes [21]

cimento, havendo atualmente os mais diversos tipos, com variações de: dimensão (número de pinos e portas lógicas); tecnologia; desempenho; custo, etc. Alguns desses dispositivos podem ser reconfigurados dinamicamente, por serem programados através de células de SRAM (*Static Random Access Memory*). A possibilidade de mudança de suas funcionalidades pode ser utilizada para reduzir a dimensão de sistemas embutidos. Um FPGA, por exemplo, pode executar duas ou mais funções num sistema, desde que apresente recursos suficientes para implementar, no mínimo, cada uma delas separadamente, e haja tempo suficiente para sua reconfiguração entre as execuções. Nesse caso, dois ou mais componentes de hardware dedicado podem ser substituídos por um único FPGA que ocupe aproximadamente a mesma área de apenas um deles.

Muitos dos sistemas embutidos com os quais interagimos devem funcionar corretamente, sob pena de nos causarem sérios danos. O uso de componentes confiáveis na implementação pode dar a tais sistemas um alto grau de confiabilidade. Isso porém não impede que, ocorrendo uma falha, seu funcionamento seja interrompido. Para evitar isso são necessários componentes reserva (ou redundantes), disponíveis para substituição em caso de falha. Esses componentes reserva podem aumentar o custo e a dimensão do sistema de maneira considerável, sem mencionar a estrutura necessária para a reconfiguração do sistema em caso de falha.

Nosso trabalho propõe o uso de FPGAs dinamicamente reconfiguráveis como forma de reduzir a dimensão e o custo de sistemas embutidos tolerantes a falhas. Nele é proposta uma abordagem na qual FPGAs atuam como recursos redundantes compartilhados, para a execução

de funções do sistema implementadas em hardware dedicado. O compartilhamento pode ocorrer tanto no espaço quanto no tempo. O primeiro ocorre pela configuração simultânea de uma ou mais funções no FPGA, enquanto o segundo é obtido pela alteração, durante a execução do sistema, das funções do FPGA, através de reconfiguração dinâmica. Quanto maior o número de funções que compartilham um mesmo dispositivo, menor a quantidade de componentes reserva no sistema e, conseqüentemente, sua dimensão e custo. Denominamos o conjunto de FPGAs dinamicamente reconfiguráveis utilizados como componentes reserva de **redundância coletiva** do sistema.

Com o objetivo de auxiliar no projeto de sistemas embutidos tolerantes a falhas empregando redundância coletiva, desenvolvemos um algoritmo capaz de fornecer: 1) uma alocação de FPGAs suficiente para atender os requisitos de redundância das funções em hardware no sistema, dentro de suas restrições de tempo real; 2) um mapeamento e escalonamento das funções redundantes em FPGAs, antes e depois da ocorrência de falhas; 3) escalonamentos das funções nos elementos de processamento do sistema, antes e depois da ocorrência de falhas.

A organização desta dissertação é descrita a seguir.

No **Capítulo 2** é fornecido um panorama geral sobre sistemas embutidos e seus projetos, especialmente no que se refere à etapa de síntese. As principais características dos algoritmos de síntese de sistema atualmente empregados são descritas e é apresentado o modelo de especificação de sistemas embutidos utilizado neste trabalho.

No **Capítulo 3** é apresentado um breve histórico dos dispositivos reconfiguráveis, dedicando atenção especial aos FPGAs. São discutidos também algoritmos de síntese que fazem uso de dispositivos reconfiguráveis.

No **Capítulo 4** são apresentados conceitos referentes ao projeto de sistemas tolerantes a falhas, junto com uma revisão sucinta de alguns algoritmos para síntese de sistemas embutidos tolerantes a falhas disponíveis na literatura.

No **Capítulo 5** o conceito de redundância coletiva é apresentado junto com uma proposta de arquitetura para sua realização. A operação de um sistema tolerante a falhas por redundância coletiva é descrita em três situações: funcionamento normal, durante falha e após tratamento de falha.

No **Capítulo 6** é apresentado o algoritmo desenvolvido em suporte ao conceito de redundância coletiva através de FPGAs.

No **Capítulo 7** são descritos exemplos de projeto usados para teste do algoritmo proposto. São apresentados 4 exemplos sintéticos e um baseado no esquema de compressão e descompressão de imagem JPEG (*Joint Photographic Experts Group*).

No **Capítulo 8** são apresentadas as contribuições deste trabalho e sugestões para sua extensão.

No **Apêndice A** são mostradas graficamente as especificações funcionais dos exemplos sintéticos empregados na avaliação do algoritmo.

Capítulo 2

Projeto de Sistemas Embutidos

Este capítulo tem por objetivo introduzir conceitos relativos ao projeto de sistemas embutidos, em especial os que se referem à etapa de síntese de sistema. Os principais tipos de algoritmo de síntese propostos na literatura são comentados e apresentados. O capítulo termina com a apresentação do modelo de especificação de sistemas embutidos empregado neste trabalho, o de grafos de tarefas.

2.1 Sistemas Embutidos: Breve Introdução

Apesar de presentes nas mais diversas aplicações, não há uma definição precisa para os chamados sistemas computacionais embutidos, ou simplesmente, sistemas embutidos. Tentativas de defini-los de maneira simples como "um sistema computacional especializado que é parte de um sistema maior ou máquina" (www.webopedia.com) e "hardware ou software de computação componente de um sistema maior e que se espera funcionar sem intervenção humana" (www.dictionary.com), pecam pela generalidade. Em [56] os autores evitam simplesmente propor uma definição, optando por evidenciarem características em comum de diversos exemplos de sistemas embutidos, visando determiná-los por meio daquelas mais freqüentemente encontradas entre eles. Telefones celulares, PDAs (*Personal Digital Assistent*), fornos de microondas, secretárias eletrônicas, controle de injeção de combustível e freios de automóveis, foram alguns dos exemplos tomados, apenas para citar os mais próximos do nosso cotidiano. As características mais freqüentemente encontradas entre eles foram: a) Funcionalidade Específica; b) Restrições de Projeto; c) Comportamento Reativo e em Tempo Real.

A primeira característica refere-se ao caráter dedicado desses sistemas, geralmente executando apenas uma função, de forma repetida. A segunda característica evidencia as restrições impostas pela necessidade de portabilidade (baixo consumo de potência, dimensão reduzida), de produção em larga escala (baixo custo) e de ambiente de operação (capacidade de processamento em tempo real). A terceira característica também remete ao ambiente de operação, especificamente às reações que os sistemas embutidos devem apresentar às mudanças ocorridas nele. O processamento de dados em tempo real é necessário para que não falhem sistemas de maior porte, aos quais servem.

2.2 Projeto de Sistemas Embutidos: Panorama Atual

Ao longo do tempo, sistemas embutidos têm sido projetados usando estratégias *ad hoc*, baseadas na experiência dos seus projetistas com produtos similares e sem o auxílio de ferramentas computacionais [22]. Tal prática tem no entanto se mostrado cada vez menos adequada, seja pelo aumento no número de componentes disponíveis para os sistemas e/ou pela maior complexidade de suas funcionalidades. Definir as funções a serem implementadas em hardware e em software pode consumir um tempo significativo no ciclo de projeto dos sistemas.

Com a evolução das ferramentas de síntese, o projeto no nível de sistema tornou-se a forma mais viável para se trabalhar com sistemas embutidos. Desde o início dos anos 90, as metodologias e ferramentas computacionais no nível de sistema têm evoluído bastante, tendo sido propostos modelos e linguagens de especificação [46, 49], ambientes de desenvolvimento [40, 25, 50], e também algoritmos para síntese [39, 14, 15].

Modelos conceituais são utilizados para organizar e definir a funcionalidade de um sistema, enquanto linguagens de especificação são responsáveis pela captura desses modelos em formato executável, capaz de ser simulado computacionalmente em ambientes de desenvolvimento [24]. Como exemplos desses ambientes podemos citar o Ptolomy [40], o SpecSyn [25] e o Artemis [50].

Ambientes de desenvolvimento geralmente apóiam-se em metodologias que visam contemplar um determinado fluxo de projeto. O SpecSyn, por exemplo, baseia-se numa metodologia de modelamento hierárquico denominada especificação-exploração-refinamento,

associada ao fluxo de projeto mostrado na Figura 2.1 [25]. Sua atuação nesse fluxo é limitada às três primeiras etapas, as quais são realizadas no nível de sistema. Uma especificação funcional é capturada e empregada na exploração automática de alternativas de projeto. Em seguida, são definidos detalhes sobre sua implementação, como quantidade de elementos de processamento e de comunicação necessários, numa etapa denominada de refinamento da especificação. Maiores detalhes sobre essa metodologia são fornecidos na próxima seção.

2.2.1 Exemplo de Fluxo de Projeto: SpecSyn

O SpecSyn auxilia no projeto de sistemas embutidos da captura de sua especificação funcional até seu refinamento. A captura da especificação do sistema é realizada pela decomposição de sua funcionalidade segundo um modelo adotado, o qual é posteriormente descrito através de uma linguagem de modo à permitir sua validação por simulação e, possivelmente, verificação. Essa etapa produz uma especificação funcional sem detalhes de implementação, como ilustrado na Figura 2.1.

Na etapa de exploração são alocados vários conjuntos de componentes, nos quais os blocos da especificação funcional do sistema são mapeados. A qualidade do projeto obtido é estimada através de métricas pré-definidas. O objetivo é encontrar a alocação e o mapeamento de melhor qualidade examinados durante esse processo.

Na etapa de refinamento a especificação inicial é transformada numa nova descrição, refletindo as decisões tomadas na etapa anterior. São mapeadas variáveis para memórias, inseridos protocolos de interfaceamento entre componentes e adicionados árbitros para os barramentos. Uma descrição mais detalhada do sistema é portanto gerada, a qual pode ser verificada por meio de simulações, como mostrado na Figura 2.1. Essa descrição é utilizada no projeto dos componentes do sistema, em software e hardware.

Se mapeado num microprocessador, o componente funcional requer síntese de software. Se particionado para hardware, deve ser empregada síntese de alto nível (comportamental), convertendo a descrição funcional numa estrutura *datapath* + controlador de estados. O resultado dessa etapa é uma descrição no nível de transferência de registro *(RTL-Register-Transfer Level)*, como pode ser observado através da Figura 2.1.

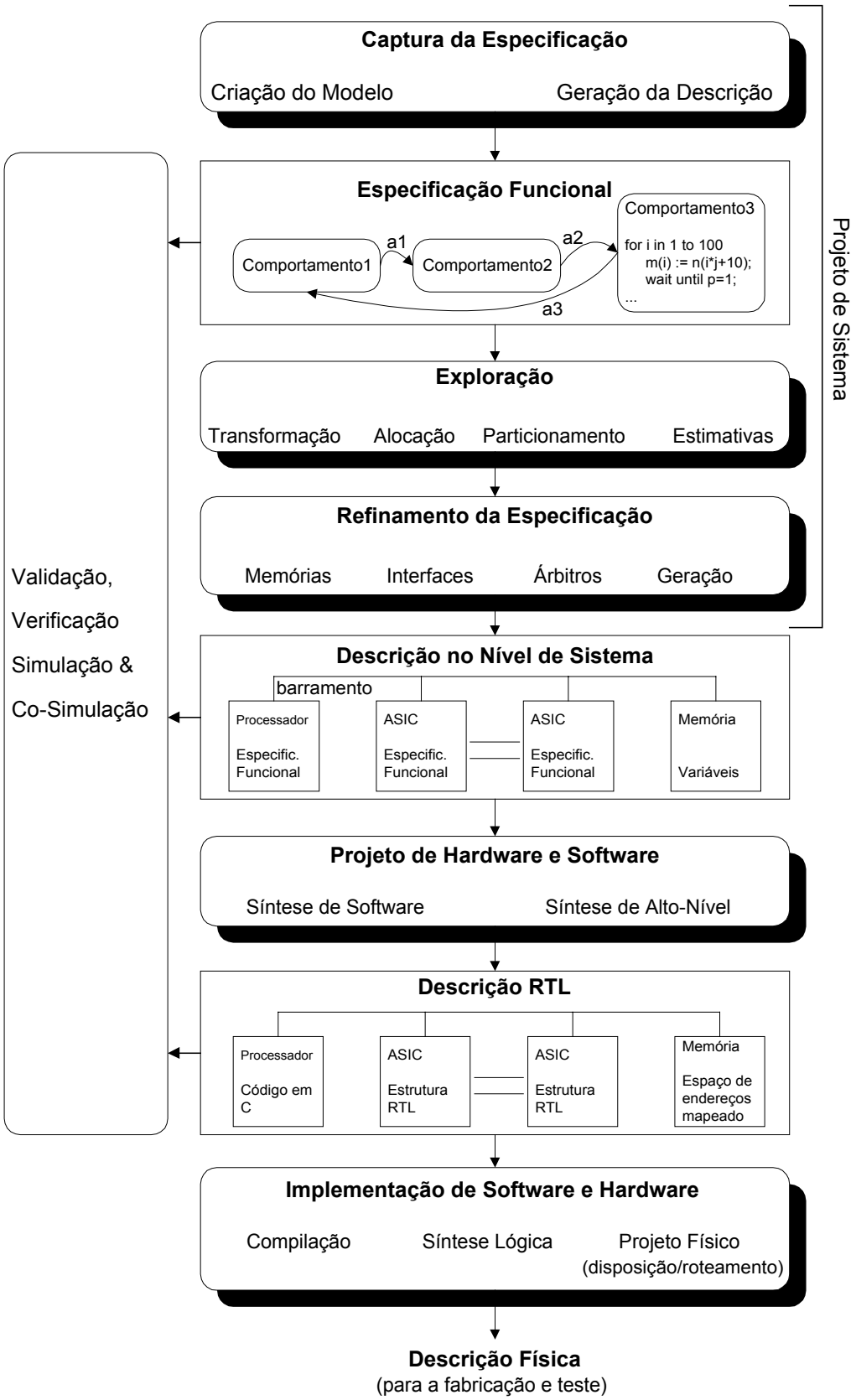


Figura 2.1 Fluxo de Projeto para Sistemas Embutidos [25]

A descrição RTL é constituída de componentes em hardware, descritos por uma HDL (*Hardware Description Language*) como VHDL (*Very high-speed integrated circuit Hardware Description Language*). Nesse nível, os componentes em software são descritos numa linguagem, como C, por exemplo. Componentes em software são compilados e componentes em hardware dedicado sintetizados, sendo gerados também seus *layouts*, possivelmente para posterior fabricação numa fundição de circuitos integrados.

2.2.2 Síntese de Sistema

No fluxo de projeto descrito na seção anterior, a etapa de exploração é realizada através de algoritmos de síntese. Dependendo do algoritmo, algumas das ações atribuídas à etapa de refinamento também podem estar incluídas no seu procedimento.

Além de uma especificação funcional do sistema, um algoritmo de síntese deve ter disponível uma biblioteca de recursos. Um conjunto de restrições de projeto também pode ser usado como entrada para o algoritmo, dependendo do modelo empregado para a especificação funcional do sistema e dos requisitos desejados para ele. Restrições de projeto podem estar incluídas na especificação funcional. Sistemas embutidos modelados por grafos de tarefas, por exemplo, têm seus prazos, que constituem restrições de tempo real, associados diretamente aos nós dos grafos. Esse modelo será detalhado mais adiante.

A biblioteca de recursos é formada pelos componentes disponíveis para alocação no projeto do sistema. A adequação dessas alocações, mapeamentos funcionais e escalonamentos, é avaliada com base nas restrições de projeto do sistema.

O procedimento de síntese é realizado em três etapas principais:

1) **ALOCAÇÃO** de elementos de processamento (PE - *Processing Element*) e de comunicação (*Links*) para o sistema;

2) **MAPEAMENTO** das funções nos PEs alocados e de seus eventos de comunicação aos *Links*;

3) **ESCALONAMENTO** das funções e eventos de comunicação de modo que nenhuma restrição de tempo real seja violada.

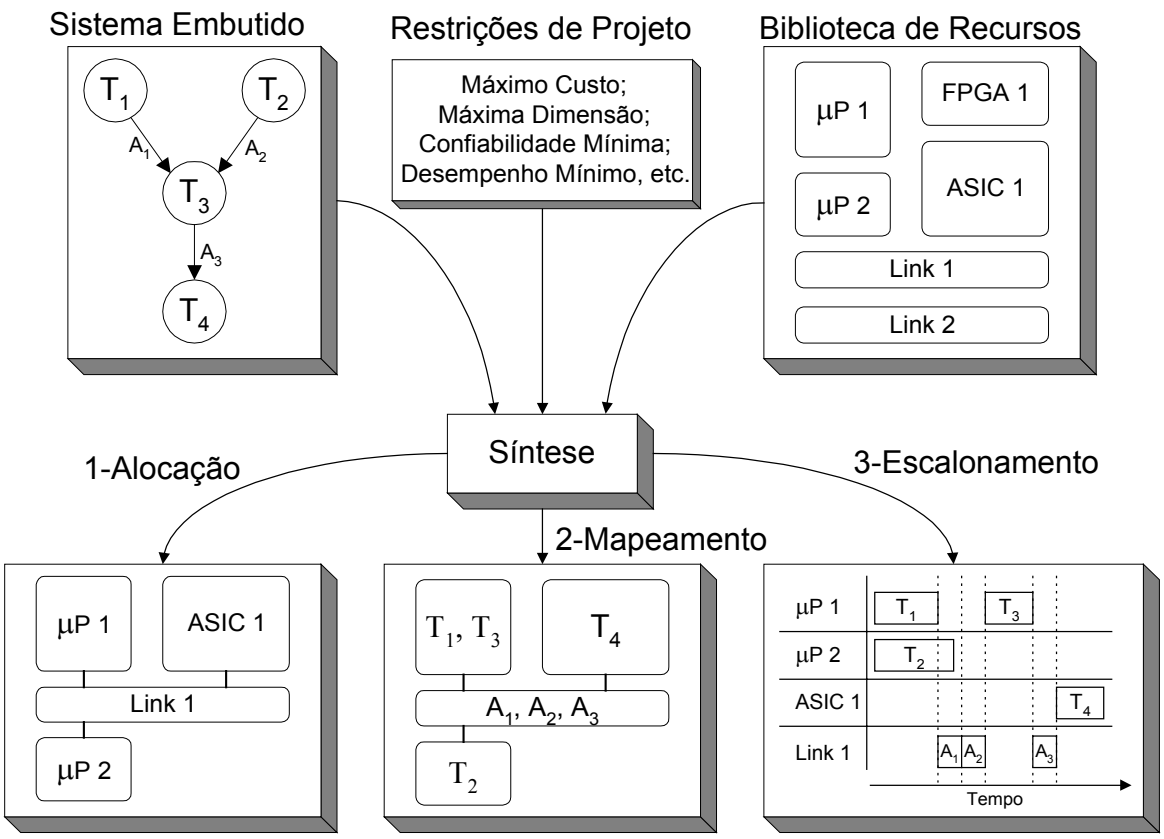


Figura 2.2 Panorama Geral para Síntese de Sistema

O objetivo da síntese é fornecer ao projetista uma ou mais descrições do sistema embutido de entrada, no nível de sistema, que respeitem as restrições e requisitos de projeto. Os requisitos geralmente são fornecidos por uma expressão denominada função objetivo, constituída de parâmetros do sistema que devem ser otimizados durante a síntese. Na Figura 2.2 são ilustradas as entradas e etapas de um algoritmo de síntese, para um sistema embutido especificado através de um grafo de tarefas.

Especificação Funcional

A especificação de um sistema embutido deve ser realizada por um modelo que capture sua funcionalidade impondo o mínimo de restrições ao procedimento de síntese empregado. Especificações baseadas em máquinas de estado podem representar de forma natural o compor-

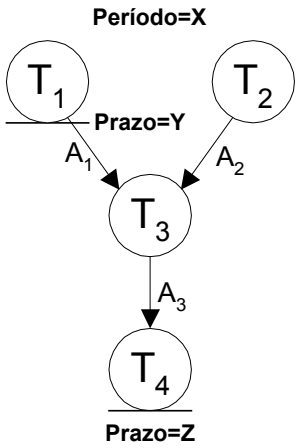


Figura 2.3 Grafo de Tarefas

tamento de sistemas reativos, no entanto apresentam o problema da explosão de estados. Para representar dois blocos funcionais concorrentes de *n* estados cada, através de máquina de estados finitos (FSM - *Finite State Machine*), são necessários n^2 estados [18], tornando proibitivo o custo computacional da síntese, sobretudo para sistemas complexos.

A especificação da funcionalidade e das restrições de tempo de sistemas embutidos através de grafos de tarefas multi-taxa é bastante utilizada, [11-20]. Grafos de tarefas são grafos acíclicos diretos (DAG - *Direct Acyclic Graph*), nos quais os nós representam tarefas (T) e os arcos (A) representam dependências de comunicação/controle. Em geral, nos trabalhos de pesquisa relativos à síntese de sistemas embutidos, assumem-se tarefas de granularidade baixa, isto é, cada tarefa[1] é suficientemente complexa para requerer várias instruções de um microprocessador [18]. Um exemplo de grafo de tarefas é dado na Figura 2.3.

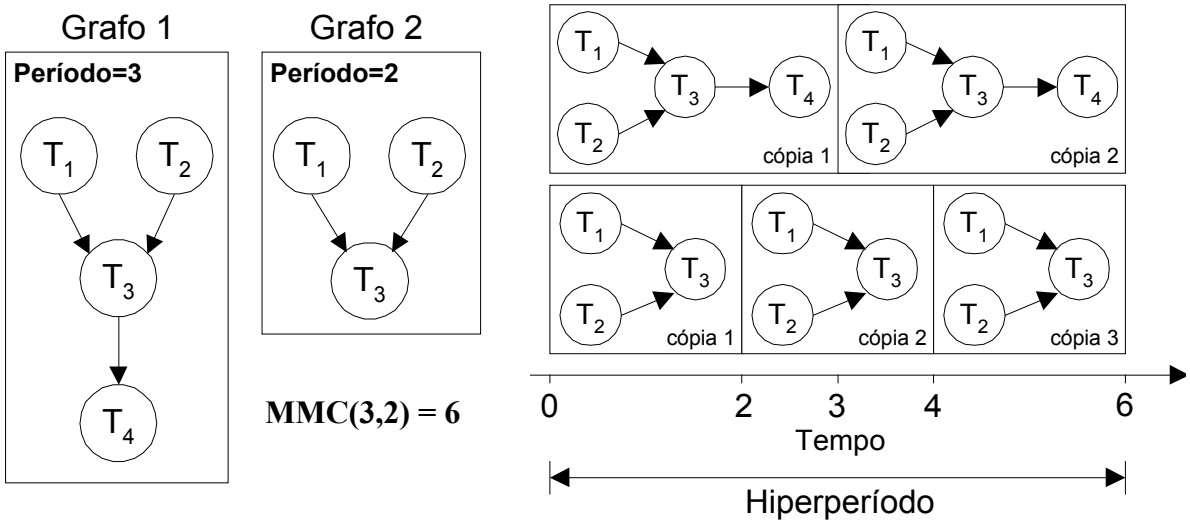
O início de execução de uma tarefa só é permitido após satisfeitas todas as suas dependências de comunicação/controle, ou seja, após todas as tarefas que se comunicam com ela através de arcos de saída terem sido executadas e os dados necessários devidamente transferidos. Além disso, seu instante de início deve ser maior ou igual ao seu EST (*Earliest Start Time*), um atributo que indica o instante mais cedo possível para a execução de uma tarefa. Por arco de saída (*resp*. entrada) entende-se o arco que sai de (*resp.* chega a) uma tarefa. O período de um grafo de

[1] Os termos Tarefa e Função são geralmente empregados para denominar blocos funcionais dos sistemas. Empregaremos no restante desta dissertação apenas o termo Tarefa, para melhor associação ao modelo de especificação funcional utilizado no trabalho, o de Grafos de Tarefas.

tarefas corresponde à menor quantidade de tempo entre o início de duas execuções consecutivas do sistema que ele representa. O prazo associado à um nó representa o máximo instante de tempo, medido em relação ao início da execução do grafo, no qual sua tarefa deve completar sua execução. Qualquer nó pode ter um prazo associado.

Sistemas Multi-Taxa

Sistemas especificados através de grafos com períodos distintos são denominados sistemas multi-taxa. Por [44], temos que a existência de um escalonamento factível para um sistema multi-taxa está condicionada a existência de um escalonamento cíclico, factível, com período igual ao Mínimo Múltiplo Comum (MMC) dos períodos dos seus grafos componentes, o qual é chamado de *Hiperperíodo*. A abordagem tradicional para tratar dessa restrição, empregada neste trabalho, consiste na replicação dos grafos ao longo do hiperperíodo, submetendo-os ao processo de escalonamento do sistema. Na Figura 2.4 o conceito é ilustrado para um sistema representado por dois grafos de tarefas com períodos distintos (Figura 2.4a). O hiperperíodo do sistema é 6, sendo mostradas na Figura 2.4b as execuções dos grafos 1 (duas) e 2 (três) ao longo dele.



a) Representação do Sistema b) Replicações dos Grafos

Figura 2.4 Conceito de Hiperperíodo

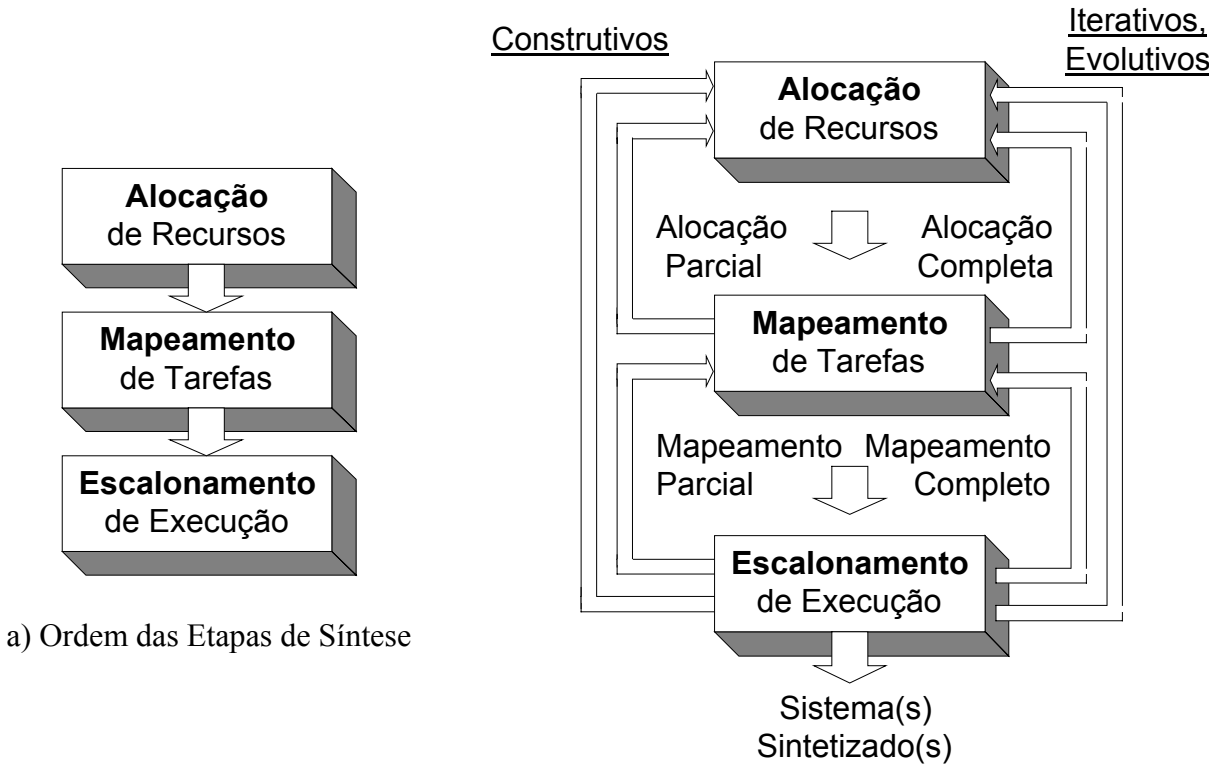
Algoritmos de Síntese

Inúmeros algoritmos de síntese de sistema já foram propostos, voltados para os mais diversos tipos de sistemas embutidos. Os voltados para arquiteturas pré-fixadas [39, 45], dispensam a etapa de alocação de recursos. Os que aplicam programação linear inteira (ILP-*Integer Linear Programming*) na etapa de mapeamento [37, 51], são inviáveis para sistemas embutidos distribuídos, devido ao elevado custo computacional inerente ao método.

O fato das etapas de alocação, mapeamento e escalonamento nesses sistemas constituírem problemas NP-completos [26], de difícil tratamento através de métodos ótimos, tem feito com que sejam tratados majoritariamente por métodos heurísticos, que embora não garantam soluções ótimas, apresentam tempos de execução reduzidos.

Recentes trabalhos em síntese de sistemas embutidos têm empregado uma das seguintes abordagens heurísticas: construtiva [11-14], iterativa (melhoria) [31] ou genética [15-17, 20, 53]. Nos algoritmos construtivos, uma solução (projeto de sistema) é obtida através de etapas nas quais lhe são agregados componentes, de acordo com avaliações parciais. Nos iterativos parte-se de uma solução completa, sendo realizadas alterações de modo a obter-se melhores valores da função objetivo. Semelhantes aos iterativos, os genéticos ou evolutivos realizam buscas não-exaustivas através do espaço de soluções do problema de modo a otimizar vários parâmetros concorrentemente. Trabalham com um conjunto de soluções, uma população [6], na qual são realizadas operações de mutação e combinação visando obter outras melhores, chamadas muitas vezes nesse contexto de "indivíduos". Tais operações continuam até que não haja melhoria por várias execuções consecutivas do algoritmo, ou seja atingido um determinado número delas, denominadas geralmente de gerações.

A seqüência alocação $\rightarrow$ mapeamento $\rightarrow$ escalonamento (Figura 2.5a) deve ser mantida, total ou parcialmente, num procedimento de síntese. Isso significa que o mapeamento de um grafo de tarefas completo, ou apenas de algum de seus nós, só pode ser realizado após a definição de uma alocação. Caso o mapeamento obtido não seja válido dentro das restrições e requisitos de projeto do sistema, a etapa de alocação pode ser realizada novamente, evitando que um mapeamento inválido seja entregue à etapa de escalonamento. Da mesma forma, a etapa de mapeamento pode ser repetida a partir da etapa de escalonamento, caso nela não seja obtido um escalonamento, total ou parcial, válido para o sistema. Há ainda a possibilidade de que o procedi-



a) Ordem das Etapas de Síntese

b) Fluxo de Informações entre Etapas

Figura 2.5 Procedimento de Síntese

mento de síntese seja refeito a partir da alocação. Na Figura 2.5b são mostradas essas possibilidades de fluxo de projeto, evidenciando que são válidas tanto em abordagens construtivas quanto em iterativas e evolutivas.

A aplicação de heurísticas complexas de otimização em cada uma das etapas de síntese não constitui uma solução computacionalmente viável, nem mesmo garante soluções ótimas. Na prática, o que se tem feito é ampliar a exploração, através de heurísticas elaboradas, em uma das etapas, aplicando nas demais métodos mais simples (para compor a solução). Em [15], baseado em heurística evolutiva, a busca genética é realizada apenas na etapa de alocação, sendo o escalonamento implementado através de uma abordagem convencional baseada em lista de tarefas prontas para escalonamento.

Avaliação dos Algoritmos de Síntese

A grande variedade de objetivos na síntese de sistemas embutidos, aliada à vasta quantidade de componentes disponíveis, tem dificultado a comparação entre algoritmos de síntese. Mesmo considerando-se aqueles com mesmos objetivos (otimização de custo sob restrições de tempo real, por exemplo) comparações são difíceis pela ausência de um conjunto de projetos padronizados (*benchmarks*), para avaliá-los. Alguns esforços têm sido realizados no sentido de facilitar a criação (e reprodução) de instâncias para o problema, que são essencialmente constituídas por uma especificação funcional e uma biblioteca de recursos. O aplicativo TGFF (*Task Graphs For Free*) [19], por exemplo, gera grafos e uma biblioteca de recursos parametrizados pelo usuário que podem ser utilizados por ferramentas que trabalham com sistemas descritos por meio de grafos de tarefas.

Entre as métricas empregadas para avaliar a qualidade do projeto de um sistema embutido destacam-se: o custo e a área ocupada pelos componentes. FPGAs dinamicamente reconfiguráveis têm sido cada vez mais empregados na minimização desses parâmetros. No capítulo seguinte algumas abordagens de síntese utilizando-os são descritas. Algoritmos de síntese para sistemas embutidos tolerantes a falhas, que trabalham com restrições e requisitos de projeto baseados em parâmetros como confiabilidade e disponibilidade, são abordados no capítulo 4.

Resumo do Capítulo 2

Neste capítulo foram introduzidos aspectos relativos ao projeto de sistemas embutidos, tendo como ponto de partida a definição de suas principais características, que os identificam e diferenciam de outros sistemas de computação. A estratégia atual de projeto de sistemas embutidos, baseada em ferramentas computacionais, foi comentada, e descrito passo a passo o fluxo de projeto associado à uma dessas ferramentas, o SpecSyn.

Foram apresentados conceitos relativos à síntese de sistema, da especificação funcional de entrada para algoritmos, até a seqüência de realização de suas principais etapas. Foi dada ênfase nos conceitos mais relevantes para a compreensão deste trabalho. O capítulo foi encerrado com comentários sobre a avaliação de algoritmos de síntese e das soluções que produzem.

Capítulo 3

Dispositivos de Hardware Reconfigurável

Neste capítulo é traçado um panorama da aplicação de dispositivos de hardware reconfigurável no projeto de sistemas embutidos. É apresentado inicialmente um breve histórico da evolução dos dispositivos lógicos programáveis (PLD – *Programmable Logic Device*), com especial atenção aos FPGAs dinamicamente reconfiguráveis. Esses dispositivos e suas características mais relevantes para este trabalho são detalhados. O capítulo é encerrado com a descrição sucinta de algumas abordagens de síntese de sistemas dinamicamente reconfiguráveis disponíveis na literatura.

3.1 Evolução dos Dispositivos Lógicos Programáveis

3.1.1 Tipos de Dispositivos

As PROMs (*Programmable Read-Only Memory*), foram os primeiros *chips* capazes de implementar circuitos lógicos por meio de programação realizada pelo usuário [10], através da utilização de linhas de endereço e dados respectivamente como entradas e saídas dos circuitos. No entanto, essa aplicação não foi bem aceita em decorrência de dois fatores: o baixo desempenho alcançado pelos circuitos, devido aos atrasos impostos pelas estruturas internas de programação das PROMs [42]; a sub-utilização do decodificador de endereços desses dispositivos na implementação de circuitos lógicos muito simples.

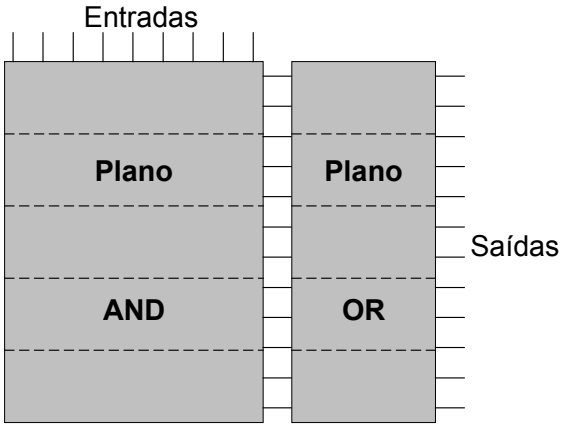


Figura 3.1 Estrutura de um PLA

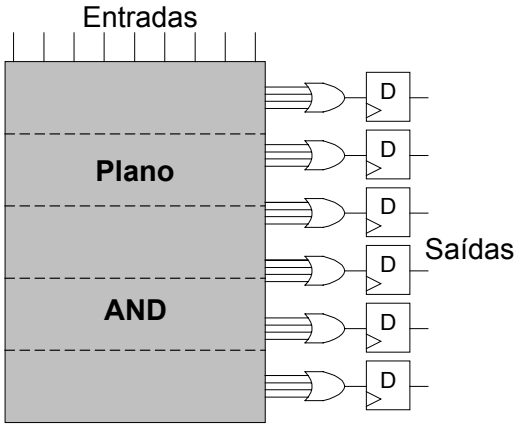


Figura 3.2 Estrutura de um PAL

Os PLAs (*Programmable Logic Array*), foram os primeiros dispositivos desenvolvidos especificamente para a implementação de circuitos lógicos [10]. Um PLA é formado por dois níveis de portas lógicas: um plano programável de portas AND, seguido por um plano também programável de portas OR, como mostra a Figura 3.1.

A estrutura dos PLAs é versátil e bastante propícia à implementação de funções lógicas na forma de soma de produtos. No entanto, na época do seu aparecimento (início dos anos 1970), seu custo de fabricação era elevado e seu desempenho bastante limitado pela presença dos dois níveis de lógica configurável. Isso motivou o desenvolvimento dos PALs (*Programmable Array of Logic*), com os quais se buscou reduzir os problemas dos PLAs através da simplificação de suas estruturas.

PALs apresentam apenas um nível programável, um plano AND, que se conecta à portas OR fixas. Diversas variações no número de entradas e saídas desses dispositivos foram fabricadas para compensar a ausência de generalidade imposta pelo plano OR fixo. Além disso, PALs contendo flip-flops, visando a implementação de circuitos seqüenciais, também foram fabricados. A estrutura de um PAL é mostrada na Figura 3.2.

Os PALs e suas diversas variações tornaram-se a base de muitas das mais sofisticadas arquiteturas de lógica programável atualmente disponíveis, sobretudo no que se refere aqueles com menores capacidades lógicas, usualmente denominados SPLDs (*Simple Programmable Logic Devices*).

O aumento da capacidade lógica de dispositivos com arquiteturas estritamente SPLD é limitado pelo rápido crescimento que ele provoca na estrutura do plano programável. Dispositivos de maior capacidade lógica baseados em arquiteturas SPLD só podem ser obtidos pela interconexão programável de múltiplos SPLDs, integrados num único *chip*. Essa é a estrutura básica de muitos PLDs atualmente disponíveis no mercado, como as famílias MAX da Altera® e XC9500 da Xilinx®, duas das maiores fabricantes de PLDs atualmente. Conhecidos como CPLDs (*Complex Programmable Logic Device*), esses dispositivos apresentam, em geral, estruturas semelhantes à mostrada na Figura 3.3.

Os dispositivos lógicos com maiores densidades de recursos programáveis são os FPGAs. Esses dispositivos foram desenvolvidos inspirados nos MPGAs (*Mask Programmable Gate Array*), matrizes de portas lógicas programadas uma única vez através de processos de fabricação numa fundição de circuitos integrados. As grandes desvantagens impostas por esse tipo de programação, ou seja, a demora na obtenção do circuito e o elevado custo inicial do projeto, foram superadas pelos FPGAs.

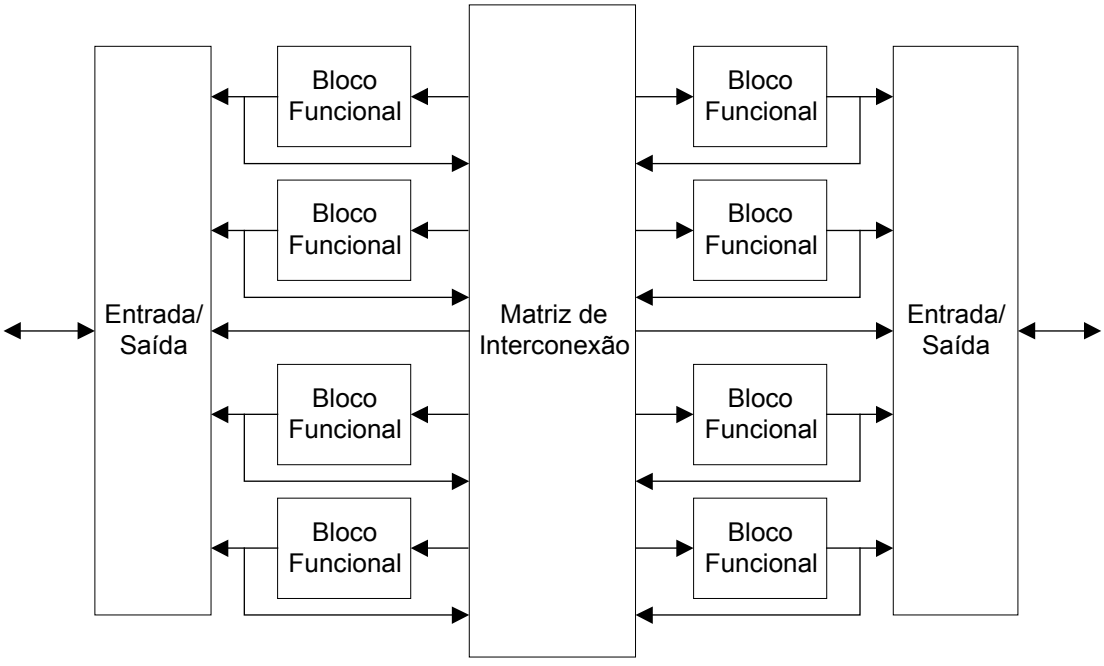


Figura 3.3 Estrutura Genérica de CPLDs

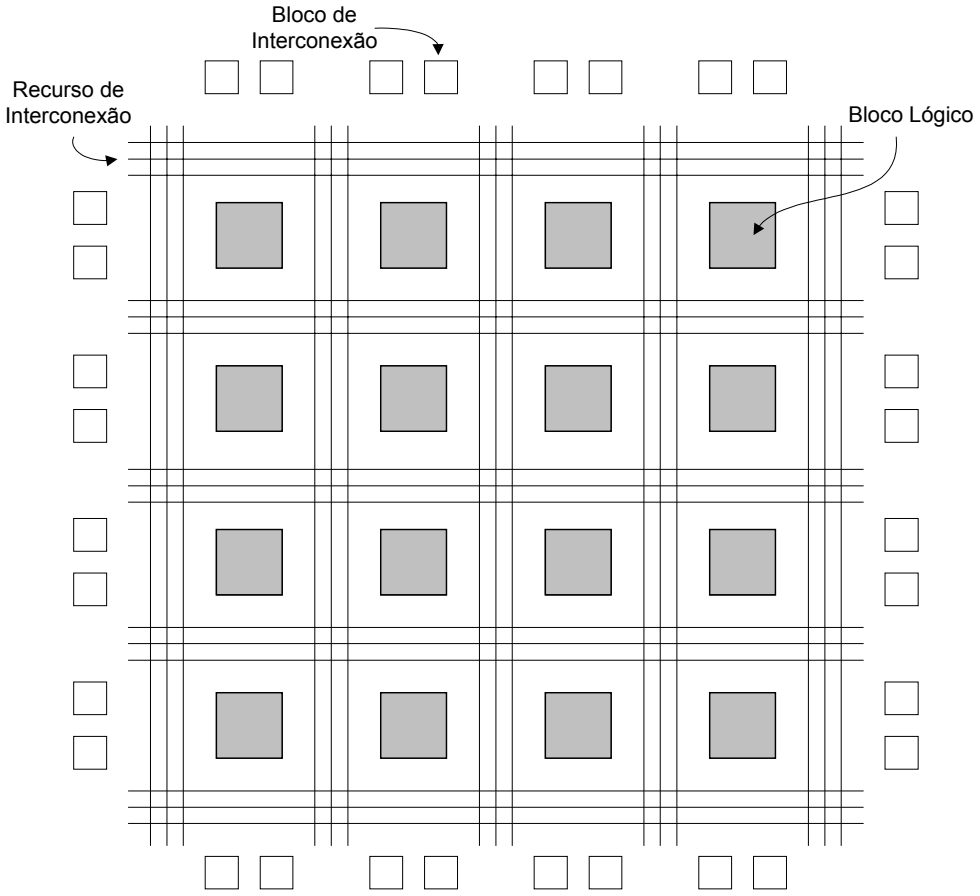
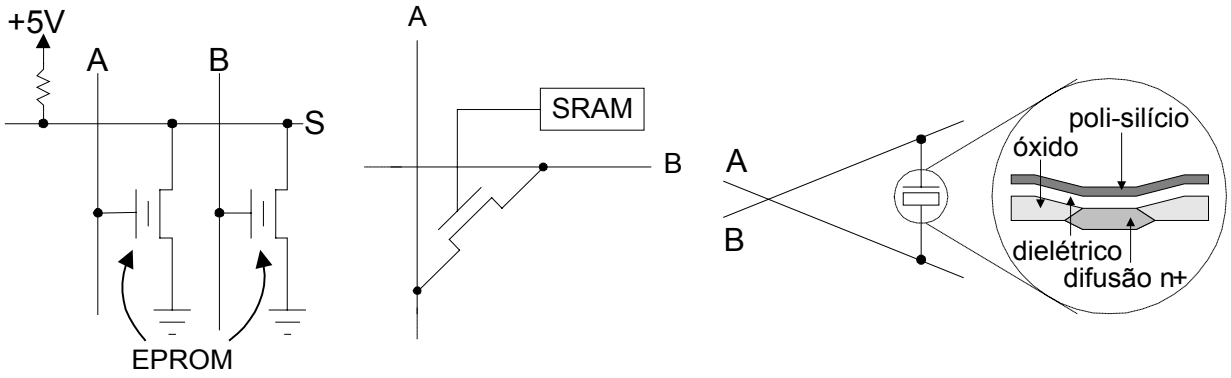


Figura 3.4 Estrutura Genérica de FPGAs

Assim como MPGAs, FPGAs consistem em matrizes de elementos, chamados blocos lógicos, e recursos para a interconexão deles. No entanto, a programação do dispositivo é realizada pelo usuário final, no "campo", e não numa fábrica. Uma arquitetura típica encontrada em FPGAs é mostrada na Figura 3.4.

3.1.2 Tecnologias de Programação

A programação de PLDs no campo é realizada através de chaves programáveis [10]. A primeira chave programável efetivamente pelo usuário foi o fusível, usado em PLAs. Embora ainda utilizado, esse tipo de programação tem sido rapidamente substituído por novas tecnologias.



a) Chave programável EPROM

b) Chave programável controlada por SRAM

c) Estrutura Antifusível da Actel®

Figura 3.5 Tecnologias de Programação: Chaves [10]

Em dispositivos de alta densidade de blocos lógicos, a tecnologia dominante é a CMOS. Nos CPLDs, a chave mais comum é implementada através de transistores com porta flutuante, como presentes em EPROM (*Erasable PROM*) e EEPROM (*Electrically Erasable PROM*). Transistores desse tipo são colocados entre dois fios para facilitar a implementação de funções lógicas, como a AND, mostrada na Figura 3.5a. Combinações nos níveis lógicos das entradas (A ou B) levam a saída (S) para nível baixo ou alto, dependendo da programação realizada nos transistores associados à elas, que os torna curtos ou circuitos abertos.

Nos FPGAs, são aplicadas majoritariamente chaves baseadas em SRAM e antifusível [10]. Chaves baseadas em SRAM são transistores que fecham, ou abrem, a conexão entre dois fios (A e B na Figura 3.5b), de acordo com a aplicação de um dado nível lógico armazenado numa célula de SRAM. Antifusíveis são circuitos abertos que, quando programados, tornam-se curtos entre fios (A e B na Figura 3.5c). No detalhe da Figura 3.5c é mostrada uma chave antifusível da Actel® constituída de três camadas: duas condutoras, no topo (poli-silício) e na base (difusão n+); e uma dielétrica (óxido-nitreto-óxido), no meio. Sua programação é realizada através da aplicação de uma tensão relativamente alta (18V) entre seus terminais, que gera uma corrente suficiente para derreter o material dielétrico e torná-lo um meio de condução entre as camadas condutoras [10].

As duas principais características relativas as tecnologias de programação dos PLDs são: reprogramabilidade e volatilidade. A primeira refere-se à capacidade de alteração do conteúdo funcional do dispositivo após sua primeira programação, enquanto a segunda refere-se à permanência da programação no dispositivo após desligada sua alimentação.

3.2 FPGAs e CPLDs

O mercado mundial de ASICs (*Application Specific Integrated Circuit*) e PLDs movimentou um total de $16,196 bilhões em 2001 e, segundo relatório da Gartner Dataquest, empresa especializada em consultoria para empresas de alta tecnologia, deverá crescer numa taxa anual de 14,7% entre 2002 e 2006, atingindo $32,185 bilhões ao final de 2006. Esse mercado inclui matrizes de portas (*gate arrays*), PLDs e projetos baseados em células padronizadas (*standard cells*), que responderam respectivamente por $2,090 bilhões, $2,625 bilhões e $11,480 bilhões em vendas, no montante movimentado em 2001. As projeções de crescimento anual para esses mercados, durante o intervalo 2002-2006, são: -16.4% (declínio) para as matrizes de portas, 18,1% para os PLDs e 17,1% para as células padronizadas [57]. A queda na parcela de mercado das matrizes de portas é decorrente da sua gradual substituição por circuitos integrados baseados em células e PLDs. O crescimento dos projetos com células é atribuído à rápida movimentação da indústria eletrônica em direção aos chamados SOC (*System-On-Chip*), enquanto o dos PLDs é associado ao aumento de suas capacidades, desempenhos e redução de seus custos [57]. A Figura 3.5 ilustra a evolução, ao longo de aproximadamente 10 anos, da capacidade, desempenho e preço do FPGA XC4000 da fabricante Xilinx®.

Pelos gráficos da Figura 3.6 é possível observar que os dispositivos, ao longo de quase uma década, tornaram-se: 20 vezes maiores; 5 vezes mais rápidos; e 50 vezes mais baratos. Essa evolução tecnológica dos FPGAs, aliada ao crescimento em suas vendas, têm gerado especulações quanto ao fim do desenvolvimento de ASICs [52].

Atualmente, PLDs são peças fundamentais em muitos tipos de sistemas. Seu uso exclusivamente em etapas de prototipagem [54] diminuiu bastante. Uma das características mais exploradas desses dispositivos tem sido a reconfigurabilidade (ou reprogramabilidade) dinâmica, isto é, a capacidade de alteração de seus conteúdos durante a operação dos sistemas nos quais

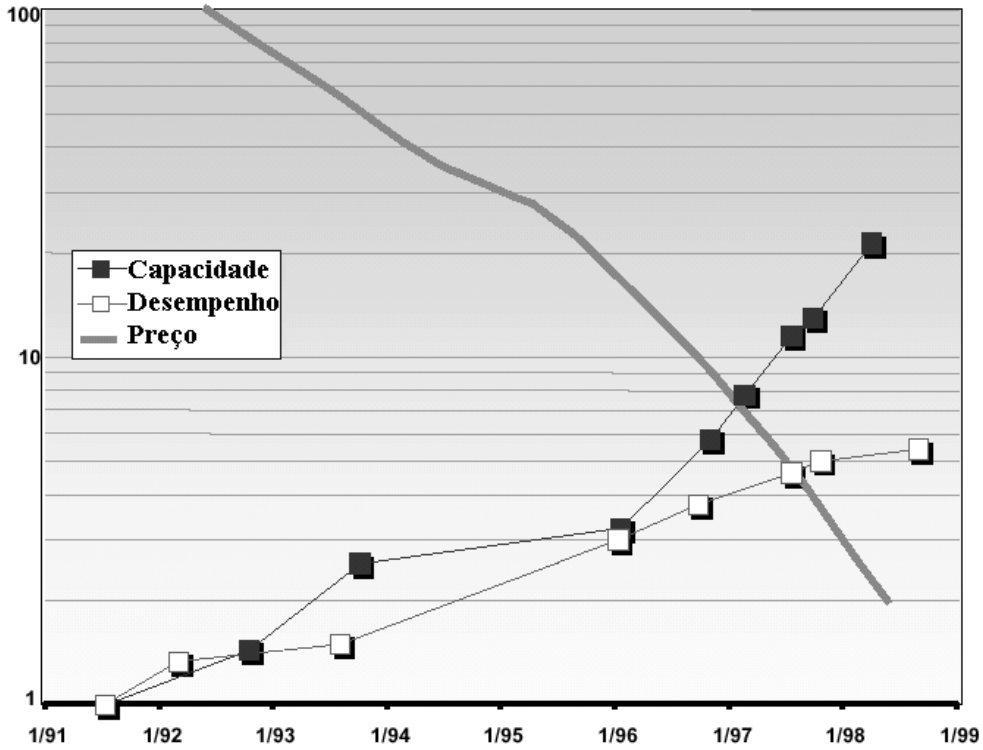


Figura 3.6 Histórico do XC4000 da Xilinx® [1]

estão inseridos [4]. Esse tipo de aplicação exige dispositivos com tecnologia de programação reprogramável e volátil. Os CPLDs e FPGAs baseados em SRAM lideram esse segmento de aplicação, sendo conhecidos como dispositivos dinamicamente reconfiguráveis.

Muitas vezes não são feitas distinções entre FPGAs e CPLDs, aplicando-se a denominação comum de FPGAs para ambos. FPGA é hoje sinônimo de lógica programável no campo.

3.2.1 Reconfiguração Dinâmica

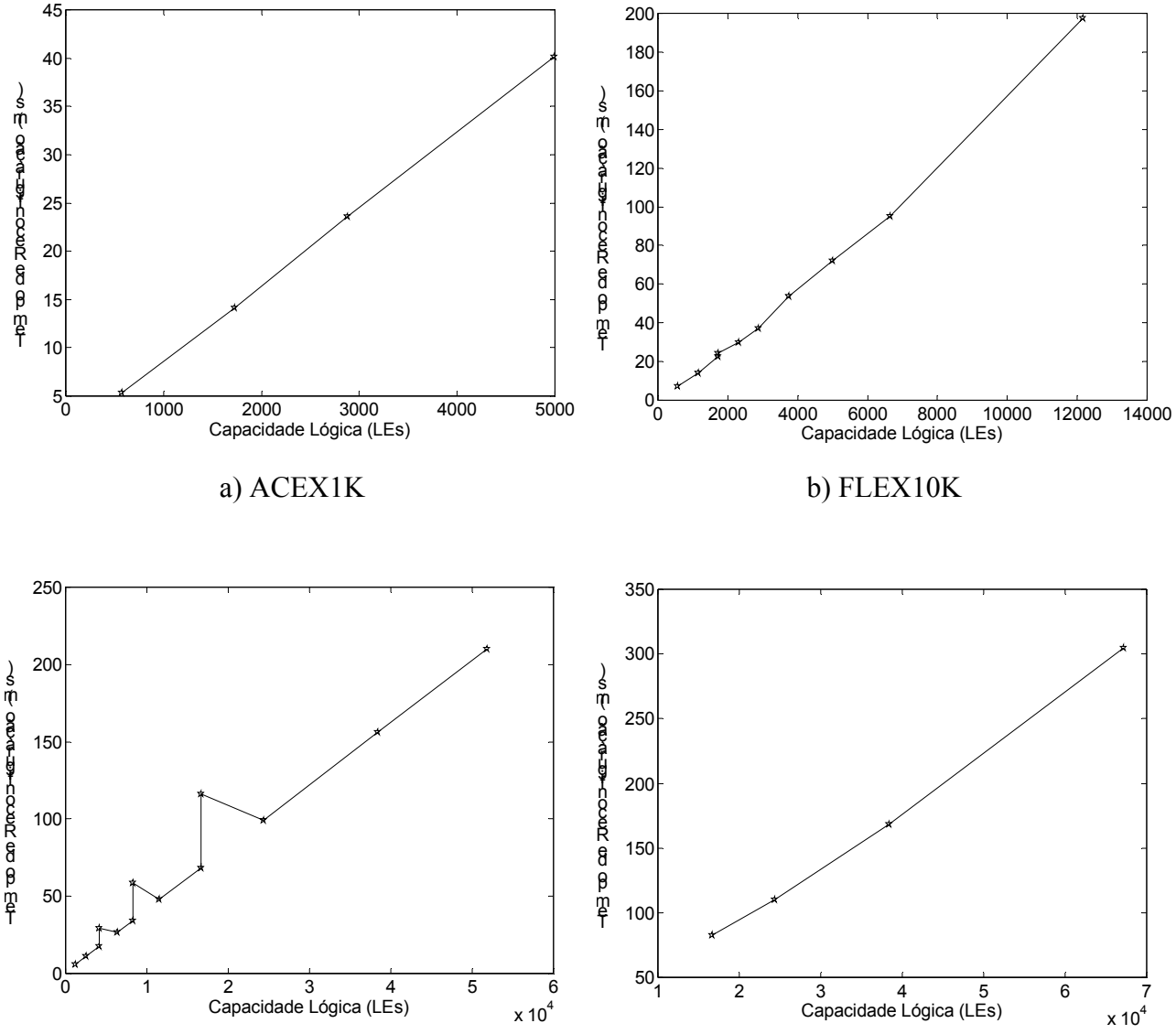
A aplicação da capacidade de reconfiguração dinâmica dos FPGAs/CPLDs tem sido defendida como forma de reduzir a dimensão e possivelmente o custo de sistemas embutidos [13, 15]. A facilidade de obtenção desses dispositivos, bem como a flexibilidade que oferecem em termos de mudanças de funcionalidade sem alteração da arquitetura do sistema, contemplam as

necessidades de um mercado no qual atualizações em produtos precisam ser realizadas em intervalos cada vez menores.

Dentro dessa nova perspectiva, há grandes apostas nos chamados co-processadores reconfiguráveis, havendo alguns desses dispositivos já prontos, tanto no meio acadêmico quanto na indústria [30, 3]. Basicamente, tais dispositivos são constituídos de um núcleo processador, com um dado conjunto de instruções, e uma área reconfigurável, disponível para ser utilizada na "construção" de um co-processador dedicado, geralmente para a aceleração de algumas operações. Esses co-processadores, integrados numa única pastilha de silício, apresentam, entre outras, as seguintes vantagens sobre sistemas hardware-software com componentes discretos: a) tempos reduzidos para transferências de dados entre hardware e software, devido ao alto grau de acoplamento existente entre essas partes; b) rápida programação da área de hardware reconfigurável, devido aos pequenos arquivos de programação, associados à instruções e não tarefas.

No que se refere aos sistemas fracamente acoplados, como os sistemas embutidos distribuídos, que trabalham com tarefas mais complexas do que simples instruções, vários problemas podem surgir devido às relações entre os tempos de execução das tarefas e os tempos de reconfiguração do FPGA utilizado. Essencialmente, esses problemas surgem pelos elevados valores de tempo de reconfiguração dos dispositivos disponíveis no mercado e pelo fato da maioria deles não suportar reconfiguração parcial, que permitiria limitar-se o tempo gasto em operações de reconfiguração pela redução na quantidade de recursos utilizados. Atualmente, apenas duas famílias de FPGAs, uma da Xilinx® (Virtex) e outra da Atmel® (AT40K) permitem esse tipo de reconfiguração. Não existe porém, ainda, ferramentas dos fabricantes disponíveis para a exploração dessa capacidade [36]. O termo reconfiguração, a menos que explicitado o contrário, será utilizado no restante deste trabalho sempre no sentido de reconfiguração total.

Nota-se, pelo acompanhamento da evolução dos FPGAs, que seus fabricantes parecem não vislumbrar o mercado de sistemas dinamicamente reconfiguráveis como sendo tão promissor quanto o mercado de SOCs com funcionalidade flexível, capaz de ser ajustada ou atualizada esporadicamente. Pode-se citar como exemplo disso os esforços no sentido de viabilizar reconfigurações através da Internet [58], nas quais o tempo de reconfiguração do dispositivo é irrelevante quando comparado ao gasto na comunicação remota de dados.



a) ACEX1K

b) FLEX10K

c) APEX20K

d) APEX II

Figura 3.7 Tempo de Reconfiguração Total x Capacidade Lógica

Enquanto os dispositivos crescem em capacidade de implementação de hardware (Figura 3.6) e o desenvolvimento de recursos IP (*Intellectual Property*) direcionados à eles se expande rapidamente [5], os tempos de reconfiguração não parecem seguir uma trajetória descendente, ou mesmo no mínimo estável. Ao contrário, os tempos têm se mantido quase diretamente proporcionais às capacidades dos dispositivos. A Figura 3.7 apresenta curvas Tempo de Reconfiguração x Capacidade Lógica para quatro famílias de dispositivos da Altera®, obtidas a partir de

dados e expressões fornecidas em [2]. As capacidades lógicas são dadas em LEs (*Logic Element*), a unidade lógica de menor granularidade nesses dispositivos. Nota-se pelas Figuras 3.7a, 3.7b e 3.7d que os tempos de reconfiguração dos dispositivos das famílias ACEX1K, FLEX10K e APEXII crescem monotonicamente com o aumento de suas capacidades. Esse comportamento também seria observado na família APEX20K (Figura 3.7b) caso tivessem sido considerados apenas dispositivos empregando um mesmo processo de fabricação. Geralmente essas diferenças decorrem do emprego de materiais distintos nas suas camadas de conexão.

A prioridade no aumento da capacidade dos dispositivos torna-se evidente quando plota-se, em escala logarítmica, pares $\left(\text{Capacidade Lógica}, \frac{\text{Tempo de Reconfiguração}}{\text{Capacidade Lógica}}\right)$ para vários dispositivos, tal como apresentado na Figura 3.8. Enquanto existe uma variação de 11.667% entre a maior e a menor capacidade plotada, a mesma variação com relação aos tempos gastos na reconfiguração de um PE[2] é de apenas 400,5%. Embora não seja mostrado, os dispositivos da Xilinx® apresentam características semelhantes.

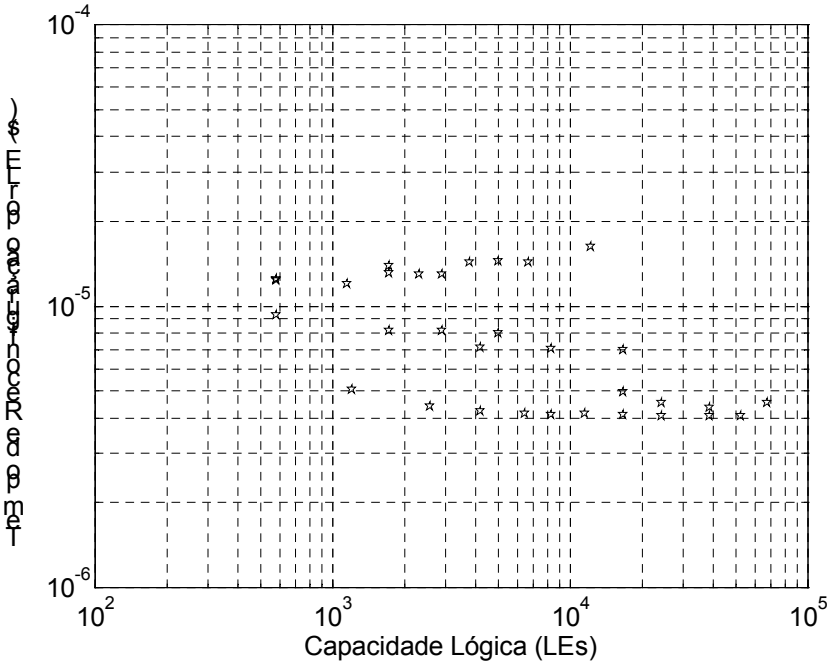


Figura 3.8 Taxa de Reconfiguração x Capacidade Lógica

[2] Esse parâmetro tem por objetivo apenas evidenciar a prioridade dada pela indústria ao desenvolvimento de dispositivos com quantidades cada vez maiores de recursos programáveis, que não se restringem necessariamente aos LEs. Muitos FPGAs têm memória embutida e seus tempos de reconfiguração englobam também sua programação.

Pode-se concluir portanto que os tempos de reconfiguração dos FPGAs são uma forte restrição no projeto de sistemas dinamicamente reconfiguráveis, pela relação atualmente existente entre eles e a quantidade de recursos programáveis nos dispositivos. Na síntese desses sistemas, a definição da granularidade de suas especificações funcionais e do tipo de FPGA utilizado são fundamentais para que se busque alternativas viáveis de projeto. Na seção seguinte são comentados alguns trabalhos de síntese de sistemas embutidos empregando reconfiguração dinâmica.

3.2.2 Síntese de Sistemas Embutidos Dinamicamente Reconfiguráveis

O emprego de hardware dinamicamente reconfigurável na síntese de sistemas embutidos é recente, tendo surgido por volta de 1998 [15]. Desde então, diversos trabalhos foram publicados nessa área, muitos dos quais incorporando características de sistemas de computação reconfigurável ao projeto de sistemas embutidos. Em geral, esses trabalhos caracterizam-se pelo emprego de co-processadores reconfiguráveis, como é o caso do ambiente de desenvolvimento Nimble [45], que automaticamente compila especificações funcionais em linguagem C, no nível de sistema, em códigos executáveis para o co-processador reconfigurável GARP. A granularidade funcional empregada no trabalho é no nível de *loops*, sendo possível a reconfiguração parcial da área programável do *chip*. Em [47] também é empregada reconfiguração parcial, sendo a síntese do sistema direcionada a uma plataforma μ-processador + FPGA, não integrada. Nesse trabalho, a especificação funcional do sistema é realizada no nível de tarefas.

Também trabalhando no nível de tarefas, os algoritmos CRUSADE [13] e CORDS [15] realizam a síntese de sistemas embutidos distribuídos. Os recursos de hardware programável são FPGAs/CPLDs de vários fabricantes, empregando esquemas convencionais de reconfiguração. Apenas no CORDS a reconfiguração parcial é empregada, não sendo porém fornecido nenhum detalhe sobre sua realização.

Um trabalho mais recente baseado no algoritmo CORDS expande e detalha melhor a síntese de sistemas embutidos distribuídos empregando reconfiguração parcial dinâmica de FPGAs [53]. Os autores desenvolvem toda a etapa de escalonamento de tarefas do algoritmo baseada na alocação de recursos programáveis de FPGAs da família Virtex, da Xilinx®. Embora

permita soluções de projeto mais precisas do ponto de vista de implementação, isso restringe a biblioteca de FPGAs a uma única família de dispositivos, reduzindo o espaço de projeto do sistema.

No nosso trabalho os FPGAs e CPLDs são reconfigurados de maneira completa, mesmo sendo passíveis de reconfiguração parcial. Esse modelo, além de facilitar a implementação do sistema obtido na síntese, visa ampliar a quantidade de dispositivos com os quais o algoritmo de síntese desenvolvido pode trabalhar, não restringindo-o a uma família específica.

Resumo do Capítulo 3

Neste capítulo foi apresentado um panorama dos dispositivos de hardware reconfigurável, desde o seu aparecimento até sua recente aplicação no projeto de sistemas embutidos dinamicamente reconfiguráveis. A primeira seção tratou de sua evolução, desde a idéia de realização de funções lógicas a partir de memórias ROM, até o surgimento dos FPGAs/CPLDs. Desses últimos foram comentadas as tecnologias de programação atualmente empregadas.

A segunda seção do capítulo foi reservada à apresentação das evoluções, de mercado e tecnológicas, dos FPGAs, com especial atenção aos passíveis de reconfiguração dinâmica. Foram apresentados, para vários desses dispositivos, os parâmetros capacidade lógica e tempo de reconfiguração, de modo à evidenciar a prioridade que seus fabricantes dão ao primeiro em detrimento do segundo. Para isso foram plotadas curvas Tempo de Reconfiguração x Capacidade Lógica para 4 famílias de dispositivos da Altera®, e mostrada, num gráfico logarítmico, a distribuição dos tempos de reconfiguração de apenas uma de suas unidades lógicas básicas em função de suas capacidades. A seção foi finalizada com a apresentação sucinta de algumas abordagens de síntese de sistemas embutidos dinamicamente reconfiguráveis.

Capítulo 4

Sistemas Tolerantes a Falhas

Neste capítulo são introduzidos conceitos relativos ao projeto de sistemas tolerantes a falhas, com ênfase nos relevantes para este trabalho. São também descritas e comentadas algumas abordagens de síntese de sistemas embutidos tolerantes a falhas disponíveis na literatura.

4.1 Conceitos Básicos

Um sistema capaz de permanecer operando corretamente na presença (ou ocorrência) de defeitos em seu hardware, ou erros em seu software, é dito um sistema tolerante a falhas [38]. Na terminologia empregada em projetos de sistemas tolerantes a falhas, os termos falha, erro e defeito têm conotações bem distintas e apresentam relações de causa e efeito entre si. Falhas provocam erros que, como conseqüência, provocam defeitos num sistema.

4.1.1 Falhas, Erros e Defeitos

São atribuídos ao termo falha defeitos físicos e imperfeições que possam ocorrer em componentes de software ou hardware do sistema. Em hardware, pode-se citar como exemplos de falhas, curtos e rompimentos em condutores. Em software, um *loop* sem condição válida de saída exemplifica uma falha das mais simples. Falhas podem ser classificadas quanto as suas durações em um dos seguintes tipos [38]: permanente, que se mantém indefinidamente até que uma ação corretiva seja realizada; transitória, que pode aparecer e desaparecer num curto intervalo de tempo; intermitente, que aparece, desaparece e aparece novamente repetidamente.

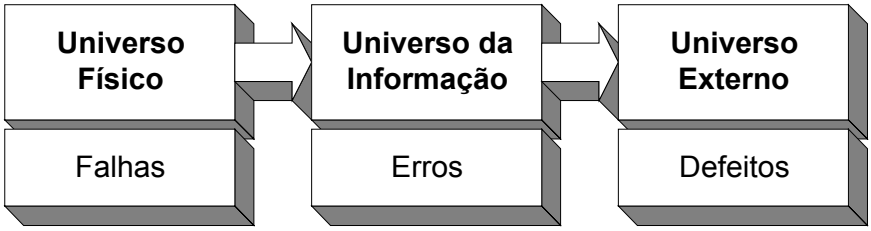


Figura 4.1 Modelo dos Três Universos [38]

Erros são manifestações de falhas que implicam em desvios na operação do sistema, provocando defeitos. Um defeito corresponde à perceptível alteração do comportamento natural do sistema.

Para auxiliar na distinção entre os conceitos de falha, erro e defeito, foi proposto em [38] o chamado Modelo dos Três Universos, ilustrado na Figura 4.1. Falhas ocorrem no Universo Físico, no qual estão contidos todos os componentes materiais dos sistemas, isto é, processadores, fontes de alimentação, memórias, etc. Erros ocorrem no Universo da Informação e são associados as alterações em unidades de informação dos sistemas, como palavras de dados e de controle. O terceiro e último universo é o Externo ou do Usuário, no qual são percebidos os efeitos de falhas e erros nos universos anteriores. É nele que se manifestam defeitos no sistema.

4.1.2 Causas de Falhas em Sistemas

Falhas podem ser causadas por problemas associados a quatro fatores relativos aos sistemas [38]: especificação, implementação, defeitos em componentes e perturbações externas.

Especificações incorretas podem levar um sistema ao defeito mesmo em situações de operação nas quais deveriam desempenhar normalmente suas funções. Especificações de temperatura de operação de um sistema fora da faixa na qual ele deve operar naturalmente, por exemplo, podem inviabilizar seu funcionamento em algumas ocasiões.

Implementações incorretas também resultam em falhas. Curtos numa placa de circuito impresso provocados durante sua fabricação e erros de sintaxe na codificação de um software são exemplos disso.

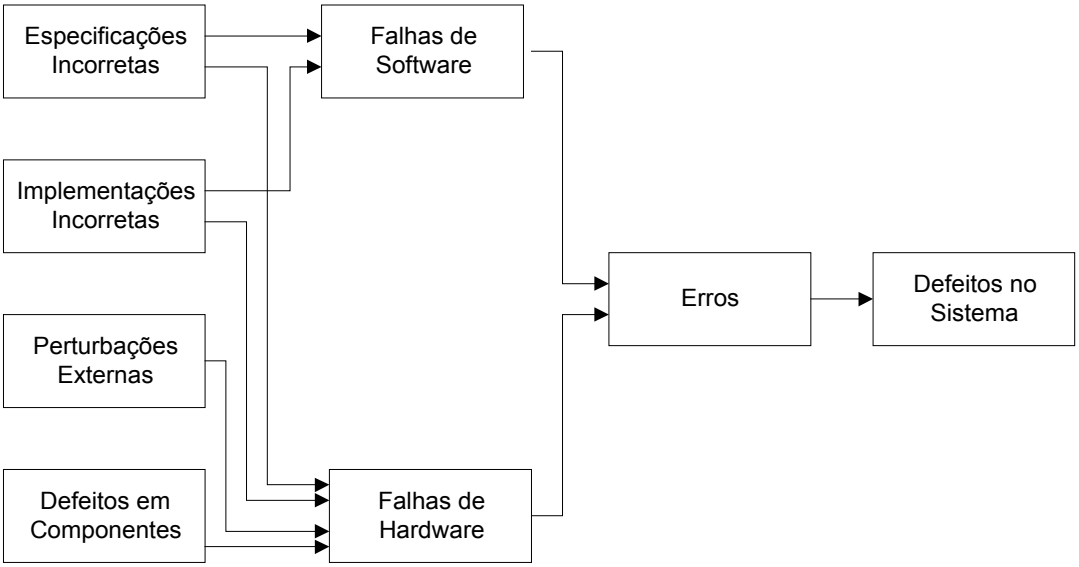


Figura 4.2 Causas de Falhas em Sistemas [38]

Defeitos em componentes são a causa mais freqüentemente associada a falhas nos sistemas e decorrem, em geral, de imperfeições em processos de fabricação ou simplesmente de desgaste por uso.

Perturbações externas incluem: mudanças, não especificadas, no ambiente de operação do sistema (variações abruptas de temperatura); interferência eletromagnética; e operação incorreta por parte de usuários. A Figura 4.2 ilustra a relação entre falhas de hardware e software e todos os fatores anteriormente citados.

4.1.3 Prevenção, Mascaramento e Tolerância a Falhas

Três abordagens podem ser utilizadas na tentativa de melhorar ou manter a operação normal de um sistema sujeito a falhas: a *prevenção de falhas*, o *mascaramento* e a *tolerância*.

Qualquer técnica que tenha por objetivo evitar a ocorrência de falhas constitui uma abordagem de prevenção. O uso de melhores componentes e ferramentas de projeto são exemplos dessa abordagem.

Técnicas de mascaramento de falhas previnem a ocorrência de erros no sistema, evitando que se propaguem através do sistema informações provenientes de falhas de software ou hardware.

Técnicas de tolerância a falhas são aquelas que possibilitam aos sistemas continuarem a desempenhar suas funções após a ocorrência de falhas, evitando portanto que apresentem defeitos.

O termo *reconfiguração* na terminologia de tolerância a falhas denota o processo de eliminação de um componente falho do sistema e sua posterior restauração para algum estado operacional. Reconfigurar um sistema requer lidar com os seguintes procedimentos:

Detecção de Falha $\rightarrow$ reconhecimento de que a falha ocorreu;

Localização de Falha$\rightarrow$ determinação de onde a falha ocorreu;

Contenção de Falha $\rightarrow$ isolamento da falha de modo a evitar que seus efeitos se propaguem pelo sistema;

Restauração de Falha $\rightarrow$ permanência ou recuperação (via reconfiguração) de um estado operacional do sistema mesmo na presença de falhas.

4.1.4 Redundância

Duas observações devem ser feitas com relação à obtenção de sistemas tolerantes a falhas [27]:

1) sempre existe a possibilidade de que um defeito venha a ocorrer no sistema se falhas são muito freqüentes ou muito severas;

2) não há tolerância a falhas sem o emprego de alguma forma de redundância no sistema.

Redundância é tudo que pertence a um sistema mas não é necessário ao seu funcionamento num estado primário, na total ausência de falhas. Ela tem, dependendo do tipo empregado, grande impacto sobre o desempenho, peso, consumo de potência, dimensão, e confiança de um sistema. Redundâncias podem ser [38]: de Hardware; de Informação; de Tempo e de Software. A seguir são comentados esses tipos, com ênfase nas técnicas de redundância de hardware mais relacionadas ao nosso trabalho.

Redundância de Hardware

A redundância de hardware consiste na replicação física de componentes de um sistema, podendo se apresentar de forma passiva, ativa ou híbrida.

Redundância de Hardware Passiva

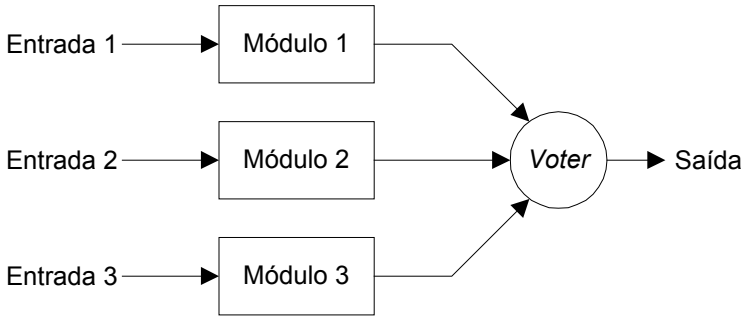
A maioria das técnicas de redundância passiva de hardware emprega o conceito de votação majoritária e torna os sistemas tolerantes a falhas sem a necessidade de detecções de falha ou reconfigurações. Dentre essas técnicas, a mais comumente empregada é a redundância modular tripla (TMR – *Triple Modular Redundancy*), que consiste em triplicar-se o hardware que executa uma dada tarefa e obter sua saída a partir de uma votação majoritária entre as saídas dos três módulos, como ilustrado na Figura 4.3a.

Se um dos módulos falha, os outros, funcionando corretamente, mascaram seu resultado falho na votação. A tarefa é portanto desempenhada de modo correto enquanto pelo menos dois módulos permanecerem operando sem falhas, e também o *voter*. Falhas em dois ou mais módulos provocam defeitos no sistema, assim como falhas no *voter*, que representa um *ponto singular de defeito*. Qualquer componente num sistema que, falhando sozinho ou isoladamente, lhe provoque defeitos, pertence ao seu conjunto de pontos singulares de defeito.

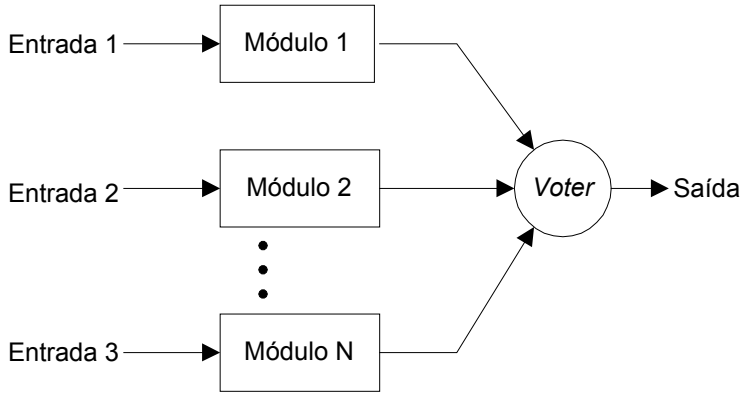
A técnica TMR pode ser generalizada na chamada redundância N-modular (NMR - *N-Modular Redundancy*), como ilustrado na Figura 4.3b. Em geral, N é um número ímpar de modo que a votação majoritária possa ser empregada. Com N módulos, um número maior de falhas pode ser tolerado. Para N=5, a tarefa permanece fornecendo um valor correto ainda que dois módulos tenham falhado.

Há diversas questões relacionadas ao *voter.* Além de ser um ponto singular de defeito em sistemas, seu sincronismo e precisão na comparação de saídas são bastante relevantes para o funcionamento correto do sistema. A escolha do tipo de implementação do *voter*, em hardware ou software, também é uma delas, relativa sobretudo à velocidade e flexibilidade[3] que se deseja obter na operação de votação. Em [41] é apresentada a implementação em FPGA de um *voter*

[3] Entende-se aqui como flexibilidade a facilidade de alteração no número de entradas do *voter*.



a) TMR



b) NMR

Figura 4.3 Mascaramento de Falhas [38]

capaz de ter sua quantidade de entradas alterada dinamicamente, para aumentar ou diminuir a capacidade de mascaramento de falhas de tarefa. Seu desempenho e flexibilidade são comparados com uma implementação semelhante realizada em software.

Redundância de Hardware Ativa

Técnicas ativas de redundância em hardware têm por objetivo a obtenção de sistemas tolerantes a falhas por meio de detecção, localização e restauração de falhas ou erros, já que a maioria das detecções são realizadas com base nos erros provocados pelas falhas. Essas técnicas

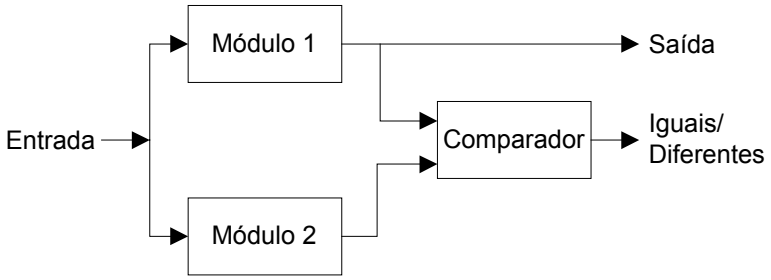


Figura 4.4 Duplicação com Comparação [38]

são comuns em aplicações que podem tolerar temporariamente resultados errôneos durante a reconfiguração do sistema. A capacidade de tolerar defeitos temporários possibilita que o mascaramento de falhas seja dispensado, o que em geral reduz o grau de redundância necessário para o sistema [38].

A duplicação com comparação é um tipo simples de redundância ativa. Dois módulos idênticos recebem uma mesma entrada e realizam sobre ela a mesma tarefa, em paralelo, comparando seus resultados e indicando quando são iguais ou não. A Figura 4.4 ilustra esse tipo de arquitetura.

Por haver a possibilidade apenas de detecções de falhas, sem localização, não há tolerância através desse esquema. Alguns problemas dessa abordagem são: livre propagação de erros, já que os módulos recebem uma mesma entrada; alta dependência do comparador, que, falhando, pode indicar falhas inexistentes nos módulos ou mesmo nunca detectá-las, que é a pior situação.

Outra técnica de redundância de hardware ativa é a reserva "em espera" (*Standby Sparing*), ilustrada na Figura 4.5. Ela consiste no emprego de um módulo de hardware executando normalmente enquanto sua saída é testada por algum esquema de detecção de erro. Se uma falha é detectada, o módulo é removido e um dos módulos reservas disponíveis passa a ser responsável pela execução da tarefa. Apenas um dos módulos é responsável pela saída do seletor, que é responsável pela reconfiguração do sistema caso lhe seja indicada uma falha no módulo em execução.

Uma vez que a reconfiguração do sistema é necessária, ele permanece defeituoso durante um certo intervalo de tempo. Para minimizar esse intervalo pode-se empregar a técnica de *hot*

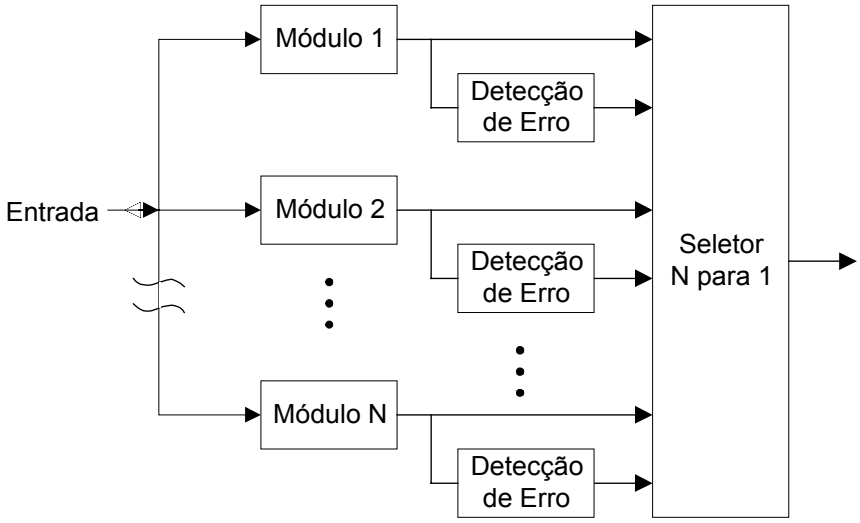


Figura 4.5 Reserva em Espera [38]

standby sparing, na qual todos os módulos reserva operam em sincronia com o módulo responsável pela saída do seletor, estando preparados para o substituírem em qualquer instante. A maior complexidade na sincronização entre os módulos e o aumento no consumo de potência do sistema são as grandes desvantagens dessa técnica.

Abordagens híbridas, como o próprio nome sugere, fazem uso de características tanto de técnicas passivas quanto ativas. Um exemplo disso é a técnica *pair-and-a-spare,* que combina duplicação e comparação com reserva em espera. Dois módulos estão sempre ativos, tendo seus resultados comparados. Detectada uma falha, o módulo que a causou é localizado (pela detecção de erro) e substituído por um módulo reserva. Uma arquitetura dessas é apresentada na Figura 4.6.

Redundância de Informações

A adição de informações redundantes aos dados que trafegam num sistema, de modo que falhas possam ser detectadas, mascaradas ou possivelmente toleradas, é a maneira pela qual é empregada a redundância de informação. Algoritmos de detecção e correção de erros em códigos formados pela inserção de dígitos de controle em palavras de dados são exemplos usuais dessa

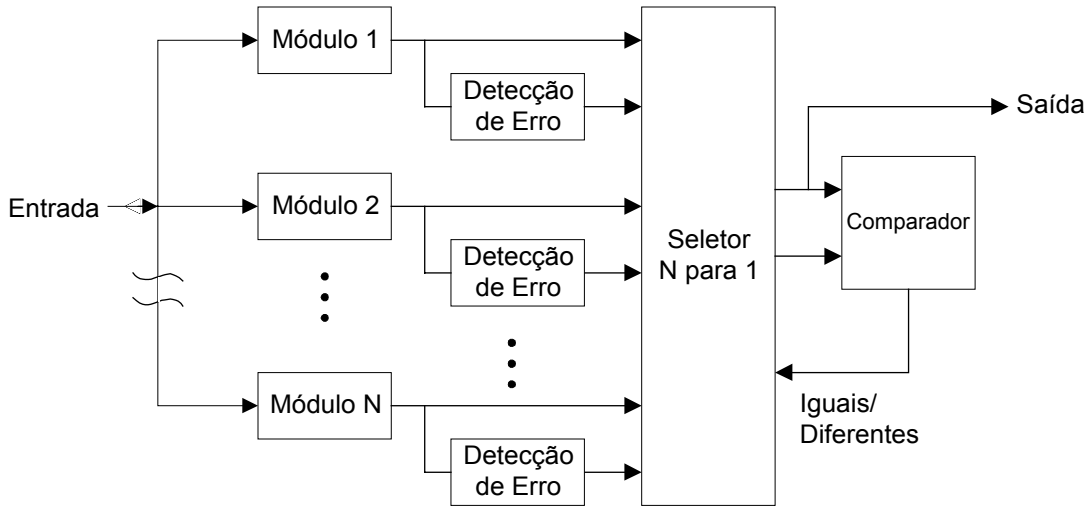


Figura 4.6 Técnica *Pair-and-a-Spare* [38]

prática, sendo alguns exemplos de códigos [38]: paridade; duplicação; *checksums*; códigos cíclicos; código de Hamming para detecção e correção de erros.

Redundância de Tempo

A redundância de tempo consiste, de maneira simples, na execução repetida de tarefas de modo à detectar falhas. Havendo diferença entre os resultados obtidos em instantes de tempo diferentes, para as mesmas entradas, um erro é sinalizado. Se o erro foi provocado por uma falha transitória, ela poderá ser detectada por um esquema como o mostrado na Figura 4.7.

Redundância de Software

Redundância de software, consiste, simplificadamente, na adição de software extra ao sistema. Entre as principais técnicas existentes tem-se a programação em N-versões (*N-version Programming*), que consiste no desenvolvimento de N versões de um mesmo software, fazendo-as operarem simultaneamente no sistema e comparando seus resultados. O princípio empregado é o mesmo da NMR para redundância de hardware, porém pelo fato de erros de software serem

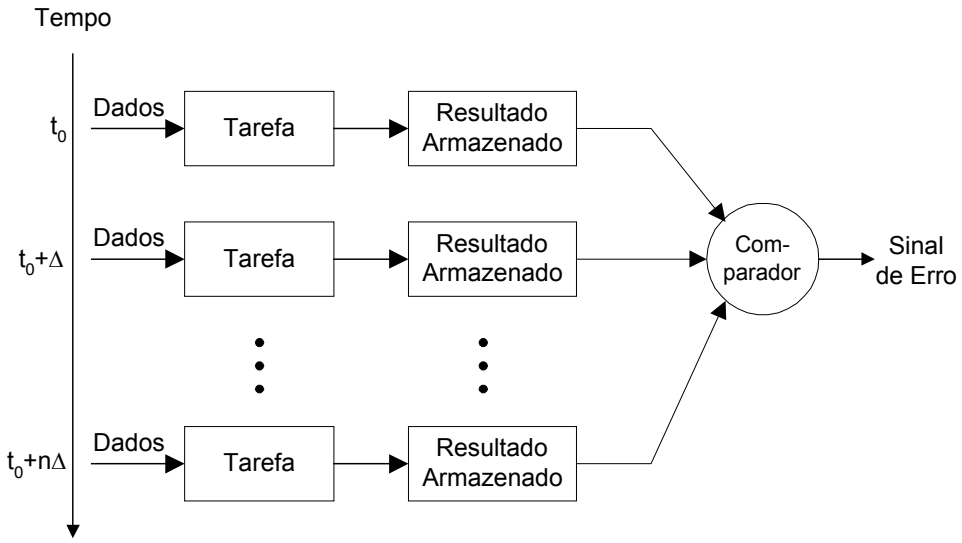


Figura 4.7 Detecção de Falhas Transitórias [38]

gerados por especificações e implementações incorretas, as versões devem ser desenvolvidas por grupos de projetistas distintos, com diferentes abordagens, visando uma mesma funcionalidade [38]. Isso diminui a probabilidade de incorreções idênticas entre as versões, no entanto aumenta os custos de desenvolvimento do sistema.

4.2 Avaliação de Sistemas Tolerantes a Falhas

Duas das características mais relacionadas ao projeto de sistemas tolerantes a falhas são: confiabilidade e disponibilidade.

Representada por R(t), a *confiabilidade* de um sistema é definida como a probabilidade condicional de que ele realizará corretamente suas funções durante o intervalo de tempo $[t_0, t]$, dado que no instante t_0 ele estava funcionando corretamente. Esse parâmetro é utilizado geralmente na caracterização de sistemas nos quais defeitos, ainda que momentâneos, são inaceitáveis. Sistemas com impossibilidade de reparo, como os embutidos em sondas de exploração espacial, também o utilizam como principal métrica de qualidade.

Pela definição de confiabilidade, é possível concluir-se que [38]: sistemas tolerantes a falhas não são necessariamente sistemas com alta confiabilidade. Falhas com grande probabilidade de ocorrência, ainda que toleradas, reduzem a confiabilidade de um sistema; Sistemas com alta

confiabilidade podem não ser tolerantes a falhas. Bons componentes elevam a confiabilidade de um sistema porém suas falhas ainda provocam defeito no sistema.

A *disponibilidade* de um sistema é representada por A(t), sendo definida como a probabilidade de que ele esteja operando normalmente e disponível para realizar suas atribuições no instante de tempo t. Esse parâmetro é usado na caracterização de sistemas cujo principal objetivo é prover seus serviços sempre que possível. Enquanto a confiabilidade é calculada sobre um intervalo de tempo, a disponibilidade é calculada num instante. Assim, um sistema pode ser altamente disponível mesmo que apresente períodos freqüentes, porém extremamente curtos, de inoperabilidade, o que necessita o emprego de rápidas ações de reparo. Terminais bancários de auto-atendimento são sistemas que requerem alta disponibilidade.

Um parâmetro também bastante empregado na caracterização de sistemas tolerantes a falhas é a *dependabilidade*. Proposto originalmente como "a *qualidade do serviço entregue por um sistema que justifica a confiança depositada nele"* [43], sua quantificação pode ser realizada através da confiabilidade e/ou disponibilidade do sistema, ou através de outros parâmetros como segurança e testabilidade [38].

4.3 Síntese de Sistemas Embutidos Tolerantes a Falhas

Em [11] é proposto um algoritmo para síntese de sistemas embutidos distribuídos tolerantes a falhas denominado COFTA. A especificação funcional do sistema é feita através de grafos de tarefas e a síntese realizada no nível de sistema. O algoritmo emprega uma abordagem construtiva e tem como principal objetivo a minimização do número de mecanismos de detecção de falhas empregados no sistema, dentro das restrições de latência de falha, confiabilidade e disponibilidade especificadas. Entre esses mecanismos de detecção de falhas encontram-se esquemas de duplicação e comparação e códigos de detecção de erros em comunicações de dados. A *latência de uma falha* é definida como o tempo transcorrido entre sua ocorrência e detecção [11]. Para reduzir o número de tarefas envolvidas na detecção de falhas, é realizado o compartilhamento dessas tarefas entre os vários módulos do sistema. Para cumprir o requisito de *cobertura de falhas* de uma tarefa, são feitas combinações entre suas tarefas de teste. Uma abordagem convencional de redundância é empregada, na qual os módulos defeituosos são substituídos por outros idênticos à eles. O sistema emprega contadores de falhas para distinguir

entre falhas transitórias e permanentes, indicando que os sistemas obtidos podem tornar-se temporariamente indisponíveis. Não são consideradas re-execuções de tarefas após falhas.

Pela descrição anterior, percebe-se que o COFTA emprega majoritariamente redundância de hardware. Há outros tipos de redundância nele, porém associados aos mecanismos de detecção (tarefas de teste), como de informação, nos códigos detectores de erros. Muitos algoritmos de escalonamento de sistemas em tempo real empregam algum tipo de tolerância a falhas. Em [29] é apresentada uma heurística para a obtenção de um escalonamento distribuído tolerante a falhas. As falhas consideradas são defeitos em processadores, mais precisamente falhas silenciosas. Isso significa que um processador quando falha deixa de interagir permanentemente com outros componentes do sistema. A tolerância a falhas é obtida através de redundância de software (nas tarefas) e de tempo (nas dependências de dados). Em [28], é apresentado um esquema para a tolerância de falhas durante a execução de tarefas em tempo real preemptivas, sendo descrito um esquema de reparo utilizado para re-executar essas tarefas na ocorrência de falhas transitórias. Em [7] os autores apresentam um algoritmo de escalonamento tolerante a falhas voltado para sistemas multiprocessadores em tempo real com prazos rígidos. A tolerância a falhas é obtida através da duplicação das tarefas. Cada tarefa escalonada apresenta uma cópia, ativa ou passiva, escalonada em um processador distinto do seu.

Resumo do Capítulo 4

Neste capítulo foram introduzidos os principais conceitos relativos ao projeto de sistemas tolerantes a falhas. Toda a terminologia necessária para o desenvolvimento do trabalho descrito nesta dissertação é apresentada ao longo das duas primeiras seções. Na primeira seção são apresentados os conceitos de falha, erro e defeito. Além disso, é dada a definição formal de redundância e apresentados alguns de seus tipos, com ênfase na redundância de hardware. Na segunda seção são definidas, sucintamente, as métricas confiabilidade, disponibilidade e dependabilidade, empregadas na medição da qualidade de sistemas com relação às suas possíveis falhas. Na última seção são comentados alguns algoritmos relacionados à síntese de sistemas embutidos tolerantes a falhas.

Capítulo 5

Redundância Coletiva

Este capítulo introduz o conceito de redundância coletiva por reconfiguração, proposto neste trabalho. É apresentada a motivação para sua utilização no projeto de sistemas embutidos tolerantes a falhas e proposta uma arquitetura para sua realização. É descrito o funcionamento desejado para um sistema implementado com essa arquitetura e introduzido o algoritmo de síntese desenvolvido para obtê-lo.

5.1 Motivação

Como explicado no capítulo anterior, o emprego de redundância de hardware implica no aumento da dimensão do sistema. Dependendo do tipo de implementação da tarefa para a qual o hardware adicional proverá redundância, o aumento da dimensão do sistema pode ser maior ou menor. Tarefas distintas, implementadas em software e executando em microprocessadores também distintos, porém do mesmo tipo, podem compartilhar um mesmo microprocessador reserva. Para tarefas distintas implementadas em hardware dedicado, esse compartilhamento de dispositivos reserva não é possível.

A flexibilidade encontrada na provisão de tolerância a falhas para tarefas em software é decorrente da programabilidade dos seus PEs reserva. Tal flexibilidade inexiste em redundância para hardware dedicado quando implementada também através de hardware dedicado. Havendo capacidade de programação dos PEs reserva dessas tarefas, o emprego de dispositivos exclusivos não é obrigatório, possibilitando sistemas de menor dimensão.

Uma alternativa para a obtenção dessa programabilidade é o emprego de software na provisão de tolerância a falhas para tarefas em hardware dedicado. Um microprocessador, por

exemplo, seria responsável pela execução, em caso de falhas, de várias tarefas em hardware. O compartilhamento desejado seria então obtido e a dimensão do sistema tolerante a falhas reduzida. Devido às diferenças existentes entre fluxos de projeto de hardware e de software, as tarefas reserva seriam versões das executadas em hardware dedicado. Isso implicaria na realização da técnica de N-versões no projeto do sistema, aumentando desse modo sua dependabilidade.

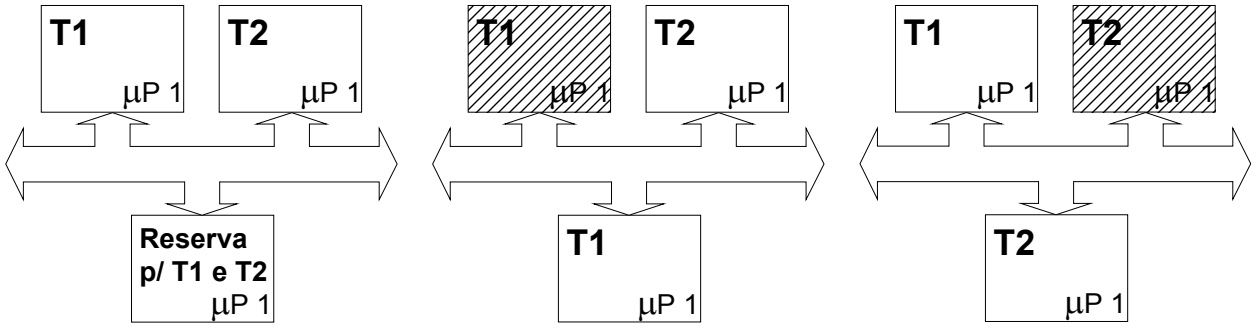
No entanto, implementações em software apresentam desempenho muito inferior às implementações em hardware dedicado. Em caso de falha, a substituição realizada, isto é, hardware dedicado por software, a menos que as restrições temporais sejam bastante folgadas, certamente implicará na impossibilidade do cumprimento dos prazos na nova configuração. Uma solução alternativa para o problema de aumento de dimensão devido aos componentes redundantes é o emprego de hardware reconfigurável, e não de software, na substituição de implementações em hardware dedicado.

O uso de hardware reconfigurável na provisão de tolerância a falhas para tarefas em hardware dedicado oferece a flexibilidade da redundância em software com desempenho próximo ao de redundância também em hardware dedicado. Para evidenciar as principais características dessa forma de redundância, considerando a necessidade de provisão de tolerância para falhas simples de um sistema composto por apenas duas tarefas, são comparados, na seqüência, as seguintes implementações para seus dispositivos redundantes: 1) em software, para tarefas em software; 2) em hardware dedicado, para tarefas em hardware dedicado; 3) em hardware reconfigurável para tarefas em hardware dedicado.

5.1.1 Tarefas em Software, Redundância em Software

A situação em que duas tarefas implementadas em software compartilham um mesmo PE reserva é mostrada na Figura 5.1a. Para evidenciar que todas as tarefas estão em software, os PEs são representados por microprocessadores (μP).

Caso o μP de T1 falhe, situação representada pelo dispositivo hachurado, sua execução passa a ser realizada pelo μP reserva, como ilustrado na Figura 5.1b. A Figura 5.1c apresenta a mesma situação para T2.



a) Sistema com Tarefas Implementadas em Software

b) Falha no μP da Tarefa T1; T1 Executando no μP reserva

c) Falha no μP da Tarefa T2; T2 Executando no μP Reserva

Figura 5.1 Situações de Falha Simples: Sistema em Software com um μP Reserva

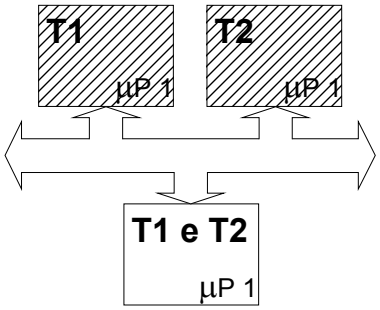
Falhando ambos μPs, como mostrado na Figura 5.2a, existe a possibilidade de que o sistema continue a operar, desde que seja possível executar as tarefas em intervalos de tempo disjuntos. Tal condição de escalonamento é apresentada na Figura 5.2b.

5.1.2 Tarefas em Hardware Dedicado, Redundância em Hardware Dedicado

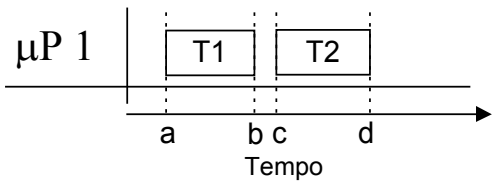
Para tarefas implementadas em hardware dedicado, são necessários PEs reservas exclusivos para cada dispositivo, como mostrado na Figura 5.3a. As Figuras 5.3b e 5.3c representam as situações de falha dos PEs de T1 e T2 e suas substituições, respectivamente.

Para o caso de falhas em ambos PEs, uma vez que existem dispositivos redundantes exclusivos para ambas tarefas, o requisito de intervalos de tempo disjuntos para as execuções de T1 e T2 é desnecessário. A situação é mostrada na Figura 5.4 e ocorre portanto sem restrições relativas ao escalonamento do sistema.

É importante observar que mesmo desejando-se tolerar apenas falhas simples do sistema, substituindo unicamente o primeiro PE a falhar, são necessários dois PEs reserva. Essa necessidade resulta numa maior capacidade de tolerância a falhas, um benefício que pode não ser interessante para o projeto, sobretudo pelo aumento provocado na dimensão do sistema. Não há



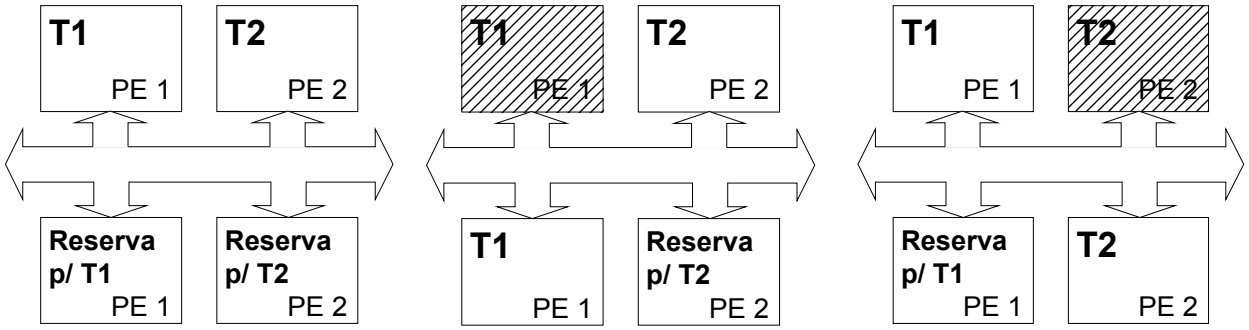
a) Falha nos μPs das Tarefas T1 e T2;
Ambas Executando no μP reserva



$$[a,b] \cap [c,d] = \varnothing$$

b) Condição para Execução de
T1 e T2 num único μP reserva

Figura 5.2 Situação de Falhas Múltiplas: Sistema em Software com um μP Reserva



a) Tarefas Implementadas
em Hardware Dedicado

b) Falha no PE da Função T1;
T1 Executando no PE reserva

c) Falha no PE da Função T2;
T2 Executando no PE reserva

Figura 5.3 Situações de Falha Simples: Sistema em Hardware Dedicado com PEs Reservas

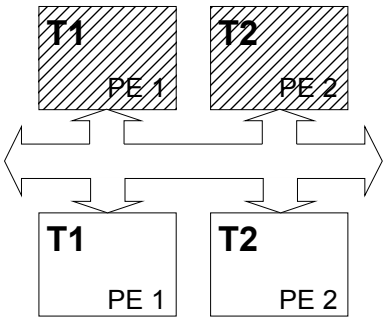


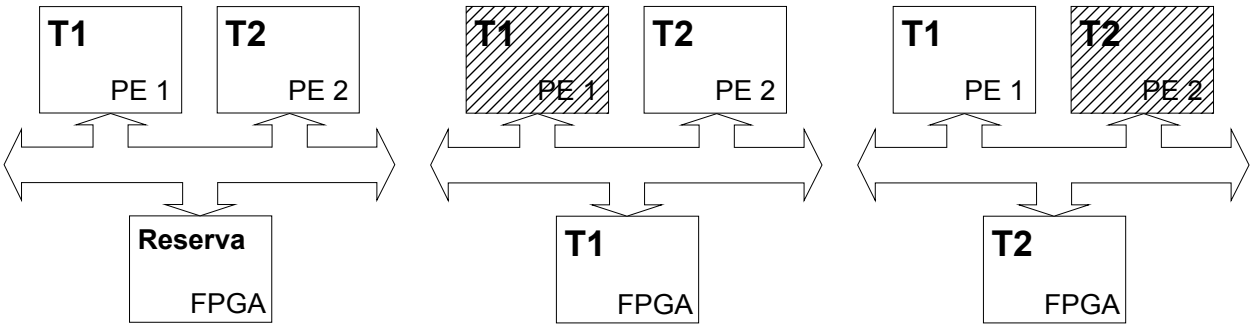
Figura 5.4 Situação de Falhas Múltiplas: Sistema em Hardware Dedicado com PEs Reservas

porém, nesse caso, flexibilidade de projeto suficiente para que a dimensão do sistema seja adequada aos seus requisitos de tolerância a falhas. O tratamento de falhas múltiplas, representado na Figura 5.4, é um "efeito colateral" decorrente do emprego de redundância em hardware dedicado para tarefas em hardware dedicado.

5.1.3 Tarefas em Hardware Dedicado, Redundância em Hardware Reconfigurável

Tomando-se um FPGA com recursos suficientes para implementar, no mínimo, versões das tarefas T1 e T2 individualmente, um sistema como o apresentado na Figura 5.5a pode ser considerado. Da mesma forma que ocorre para a redundância em software, as situações de falha mostradas nas Figuras 5.5b e 5.5c também podem ser toleradas, havendo porém a possibilidade de que o dispositivo necessite ser configurado antes de assumir a execução da tarefa.

Reconfigurações de FPGAs ainda representam tempos significativos com relação à execução da maioria das tarefas presentes em sistemas embutidos. O fato do FPGA utilizado como redundância estar ou não pronto para entrar em operação após a ocorrência de uma falha afeta significativamente o tempo de reconfiguração do sistema. Como explicado no capítulo 4, esse tempo corresponde às operações de substituição do dispositivo falho e de restauração do funcionamento do sistema. Durante a reconfiguração, o sistema permanece indisponível.



a) Tarefas Implementadas em Hardware Dedicado

b) Falha no PE da Tarefa T1; T1 Executando no FPGA

c) Falha no PE da Tarefa T2; T2 Executando no FPGA

Figura 5.5 Situações de Falha Simples: Sistema em Hardware Dedicado com FPGA Reserva

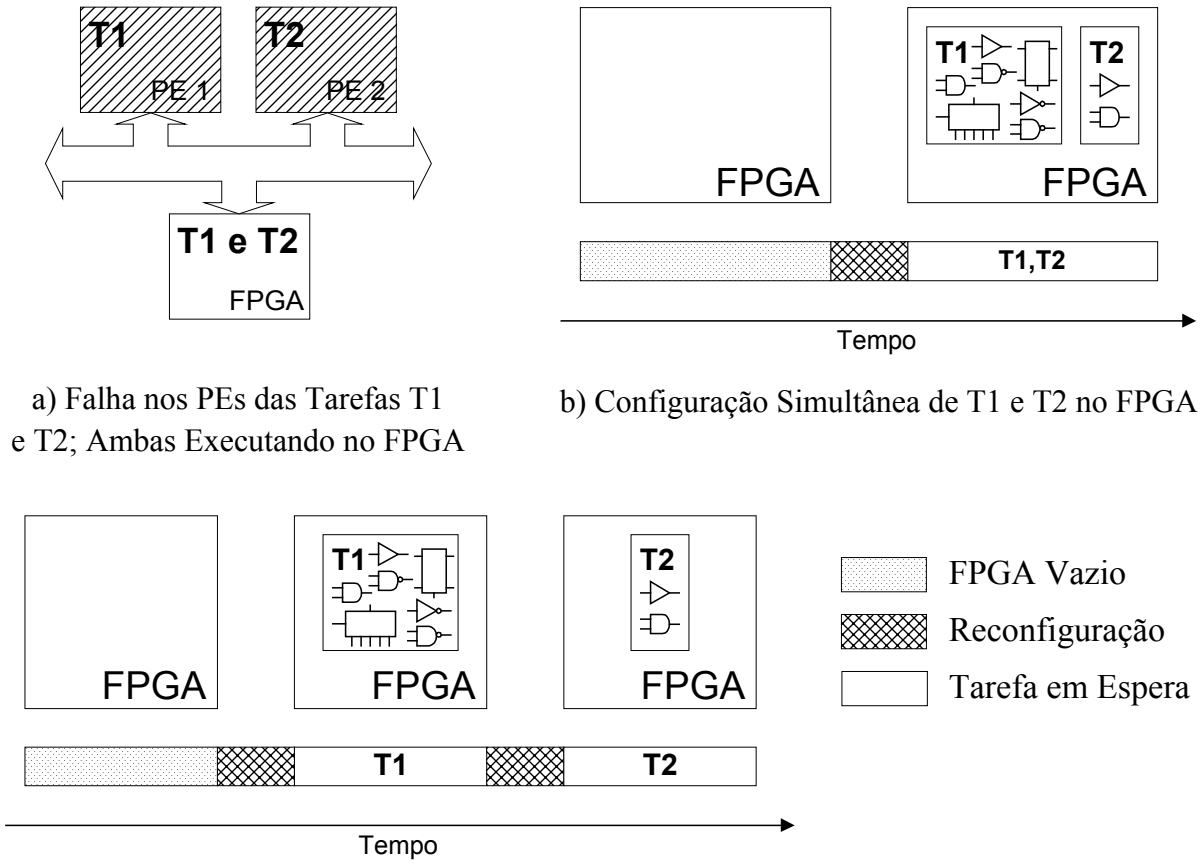
Numa situação de falha na qual a reconfiguração do FPGA seja necessária, o sistema pode ficar indisponível por um intervalo de tempo demasiado longo, comprometendo bastante seu funcionamento. Evitar reconfigurações do FPGA é portanto uma maneira de reduzir o tempo de indisponibilidade do sistema e, conseqüentemente, viabilizar o emprego de FPGAs como elementos redundantes para um conjunto maior de sistemas embutidos. A aplicação dessa técnica no nosso trabalho será melhor detalhada na seção seguinte.

Falhas em ambos PEs, como mostrado na Figura 5.6a, podem ser toleradas desde que uma das condições a seguir seja satisfeita:

1) **As versões de T1 e T2 podem ser configuradas simultaneamente no FPGA**. Nesse caso, é necessária apenas uma reconfiguração do dispositivo, o qual permanece aguardando a falha de um dos PEs para substituí-lo durante o funcionamento normal do sistema. A Figura 5.6b ilustra essa operação no tempo, com o dispositivo inicialmente sem configurações (pontilhado), durante a reconfiguração (hachurado) e em estado de espera (branco), já configurado. A situação torna-se semelhante à que ocorre com redundância através de hardware dedicado, porém, devido as possíveis diferenças entre os tempos de execução das tarefas em hardware dedicado e em FPGA, a substituição de PEs implica também na alteração do escalonamento do sistema.

2) **As versões de T1 e T2 podem ser escalonadas em intervalos de execução disjuntos, separados por um intervalo de tempo no qual o FPGA pode ser reconfigurado**. Não havendo recursos para que as versões de T1 e T2 possam ocupar simultaneamente o FPGA, o escalonamento do sistema na situação de falha de ambos PEs deve ser realizado de forma que suas execuções estejam separadas por, no mínimo, um intervalo de tempo igual ao gasto na reconfiguração do dispositivo. A Figura 5.6c mostra como deveriam estar dispostas ao longo do tempo as operações de reconfiguração do FPGA para garantir o tratamento de falhas em T1 e T2. O dispositivo, inicialmente sem configurações, é configurado primeiro com T1 e em seguida com T2.

É importante lembrar que em nenhuma das duas situações apresentadas (5.6b e 5.6c) há o compartilhamento simultâneo de recursos do FPGA entre duas ou mais tarefas, isto é, num instante de tempo, um conjunto de recursos programáveis é dedicado à execução de apenas uma ta-



a) Falha nos PEs das Tarefas T1 e T2; Ambas Executando no FPGA

b) Configuração Simultânea de T1 e T2 no FPGA

c) Configurações de T1 e T2 no FPGA

Figura 5.6 Situação de Falhas Múltiplas: Redundância Coletiva

refa. Na primeira situação, os recursos são particionados entre T1 e T2 de modo que possam operar concorrentemente. Na segunda, as tarefas dispõem individualmente de todos os recursos do FPGA, os quais são utilizados de acordo com o mapeamento definido para o dispositivo na configuração realizada.

5.2 Proposta de Arquitetura para Emprego de Redundância Coletiva

Como mostrado na seção anterior, a reconfigurabilidade dos FPGAs é fundamental para seu emprego como redundância coletiva. É necessário portanto que esses dispositivos possam ser

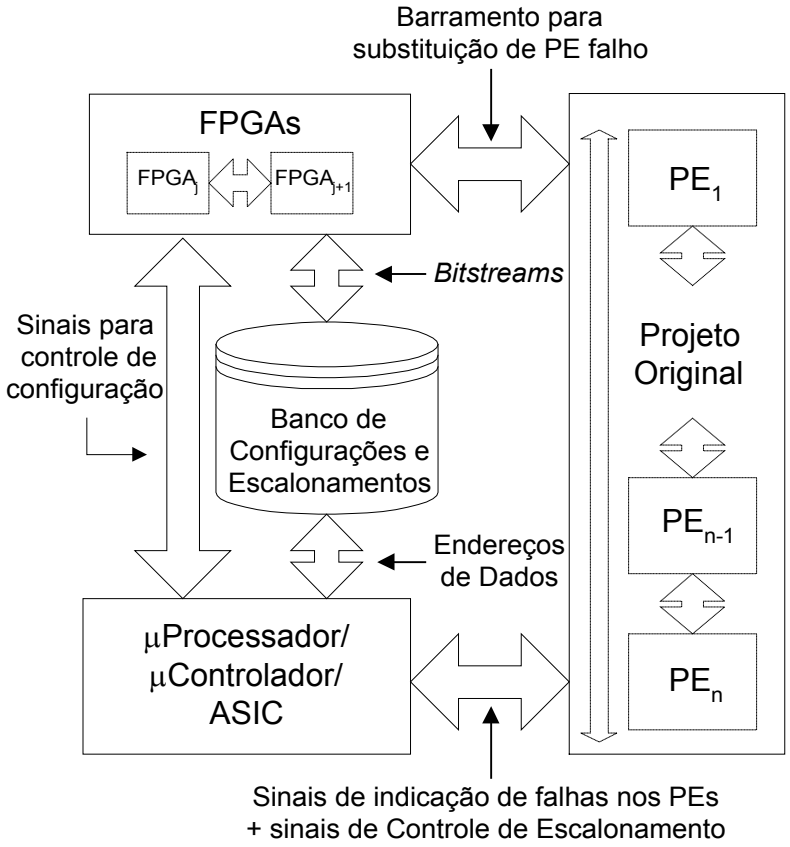


Figura 5.7 Arquitetura para Sistemas Embutidos Empregando Redundância Coletiva

submetidos à reconfigurações sempre que preciso, as quais devem ser realizadas dentro de um dos esquemas disponíveis para o tipo de dispositivo empregado [2, 59, 60]. Dentre os diversos esquemas, o mais utilizado é a reconfiguração passiva serial. Nele, um dispositivo é responsável pela transferência serial dos bits de programação (*bitstreams*) requeridos, armazenados em memória, para o FPGA. Esse dispositivo pode ser um microprocessador, um microcontrolador, ou até mesmo um circuito dedicado que implemente o protocolo de configuração do FPGA empregado.

Optou-se por propor uma arquitetura de sistema embutido tolerante a falhas na qual os FPGAs que atuam como redundâncias coletivas sejam reconfigurados através de um esquema passivo serial, tal como mostrado na Figura 5.7. O dispositivo mestre da configuração (μProcessador/μControlador/ASIC) é responsável não apenas pelas reconfigurações dos FPGAs, mas também pela reconfiguração do sistema. Em caso de falha, um sinal é emitido para que o FPGA

seja inserido no escalonamento do sistema embutido, executando a tarefa do PE falho. A substituição implica na alteração do escalonamento, devido às prováveis diferenças de tempo de execução entre as versões da tarefa em hardware dedicado e em FPGA. O novo escalonamento é lido da memória pelo dispositivo mestre, de acordo com a situação de falha sinalizada, e acionado em seguida.

Na Figura 5.7 chama-se de projeto original a alocação do sistema embutido para o qual emprega-se tolerância através de redundância coletiva. Essa alocação, bem como o mapeamento das tarefas do sistema, são dados de entrada para o algoritmo de síntese desenvolvido nesse trabalho, descrito no próximo capítulo. A Figura 5.7 tem por objetivo ilustrar, de forma simples, os elementos necessários para a realização do sistema embutido tolerante a falhas através de redundância coletiva. Essa arquitetura foi definida de modo a facilitar a compreensão da operação do sistema. Não foram realizados maiores estudos para sua implementação por não ser esse o objetivo do presente trabalho.

5.3 Funcionamento de Sistemas Embutidos com Redundância Coletiva

O funcionamento de sistemas embutidos empregando redundância coletiva baseia-se nas seguintes hipóteses:

1) Todos os PEs capazes de serem substituídos por FPGAs são módulos de hardware dedicado, executando apenas uma tarefa do sistema. Essa exclusividade, decorrente da natureza dedicada dos dispositivos, facilita a substituição dos PEs e a re-execução de suas tarefas.

2) É função do PE a detecção e indicação de sua falha. Admite-se que os PEs que necessitam de tolerância a falhas apresentam sistemas de detecção de falhas geralmente acoplados à eles. A técnica de detecção empregada não é relevante, podendo ser qualquer uma das descritas no capítulo 4. O importante é que sejam conhecidos os pontos ao longo das execuções das tarefas nos quais são indicados os resultados dos testes, de modo que a partir deles o sistema seja reconfigurado ou não. Denominados *Pontos de Detecção de Falhas* (PDF), representamos

$T_i \rightarrow$ Tarefa i

$e_i \rightarrow$ Intervalo de Execução de Ti

p_{i1}, $p_{i2} \rightarrow$ Valores Percentuais com Relação à e_i (a partir do seu início)

$\downarrow \rightarrow$ Representação de um PDF

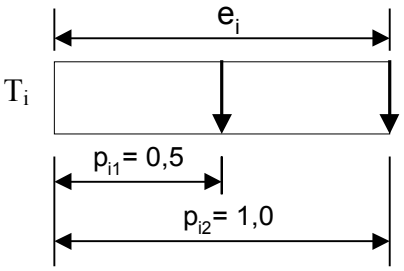


Figura 5.8 Pontos de Detecção de Falhas

sua posição ao longo do intervalo de execução da tarefa por setas verticais orientadas para baixo, tal como na Figura 5.8. Esse modelo aproxima-se bastante da realidade, já que as detecções de falhas são operações discretas no tempo.

3) Falhando um PE com redundância, sua tarefa é re-executada integralmente num FPGA. O requisito de re-execução de tarefas devido à falhas cria a necessidade de armazenamento dos dados de entrada dos PEs com redundância. Assim, devem ser empregados *buffers* capazes de conter esses dados durante o intervalo de tempo em que falhas possam ser sinalizadas. No caso da tarefa Ti da Figura 5.8, suas entradas devem permanecer durante toda a sua execução, já que uma falha pode ser sinalizada até o PDF p_{i2}. Caso existisse apenas p_{i1}, após sua ocorrência os dados de entrada não seriam mais necessários no *buffer*, pois mesmo havendo falhas, elas não seriam detectadas e, conseqüentemente, não poderiam ser toleradas. A Figura 5.9 ilustra a presença desses *buffers* juntos aos PEs do sistema.

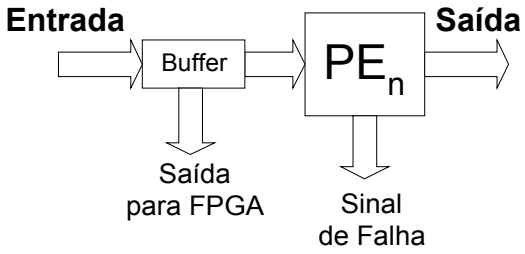


Figura 5.9 Arquitetura *buffer* de entrada + PE

4) Falha, no máximo, um PE por hiperperíodo do sistema embutido. Essa consideração é bastante empregada em sistemas tolerantes a falhas, de modo a descrever suas transições de estado de operação ao longo do tempo, sobretudo na avaliação de características como confiabilidade e disponibilidade através de cadeias de Markov [38].

Com base nessas hipóteses, dividimos o funcionamento do sistema tolerante a falhas empregando redundância coletiva em três situações de operação: normal, durante falha e após tratamento de falha. Cada uma delas é descrita a seguir.

5.3.1 Funcionamento Normal

Durante a operação normal do sistema, todas as tarefas com necessidade de redundância executam em PEs exclusivos, enquanto os FPGAs aguardam um sinal de falha para substituí-los. Como a re-execução de tarefas em caso de falhas é um dos requisitos do projeto, é necessário que o FPGA substitua o mais rápido possível o dispositivo defeituoso. O menor tempo de substituição ocorre quando o FPGA está pronto para executar a tarefa, isto é, quando sua funcionalidade já está configurada. Em muitos casos (sistemas), essa pode ser a única forma para que a re-execução de tarefas seja empregada sem violar, ou pelo menos violando o mínimo possível, os prazos do sistema, durante o hiperperíodo de execução no qual a falha ocorre. Os FPGAs devem portanto estar configurados com as tarefas nos instantes em que falhas em seus PEs podem ser detectadas, isto é, nos PDFs. Caso não seja sinalizada uma falha, a permanência da funcionalidade da tarefa no FPGA é desnecessária. Mesmo ocorrendo uma falha imediatamente após o PDF, sua detecção só poderá ser realizada por um PDF posterior, se existir, ou na próxima execução da tarefa. Isso torna maiores os intervalos disponíveis para reconfiguração dos FPGAs, permitindo uma maior flexibilização do uso de seus recursos. O FPGA funciona portanto de modo bastante semelhante ao dispositivo chamado reserva em espera, apresentado no capítulo anterior, com a configuração prévia do dispositivo sendo empregada para reduzir o tempo de reconfiguração do sistema, em caso de falhas.

Na Figura 5.10 é apresentada a situação de operação normal do $FPGA_j$ durante o intervalo de execução de uma tarefa (ou PE) para a qual ele funciona como reserva em espera.

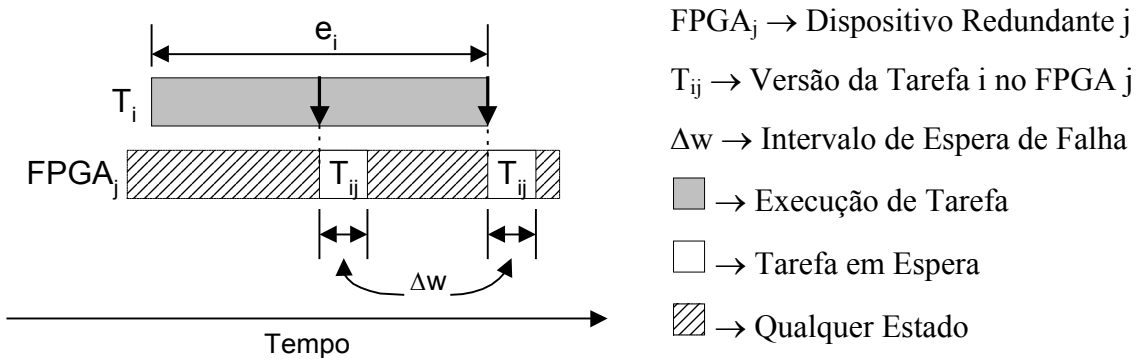


Figura 5.10 Operação Normal do Sistema

Paralelamente à execução da tarefa T_i, mais precisamente dos seus PDFs, a versão de T_i para o dispositivo $FPGA_j$, T_{ij}, permanece configurada nele. Um intervalo de tempo de espera, Δw, é empregado para que T_{ij} possa substituir T_i caso seu PE falhe. Esse intervalo é necessário para que o FPGA receba a indicação de falha do PE e suspenda uma eventual reconfiguração que venha a alterar seu conteúdo e o impeça de re-executar a tarefa T_i. Um PDF que ocorre enquanto sua tarefa correspondente está configurada num FPGA é dito **coberto** (por esse FPGA).

Na Figura 5.10, consideramos o escalonamento do $FPGA_j$ irrelevante nos intervalos hachurados para evidenciar a importância da configuração de T_{ij} apenas durante os PDFs de T_i e seus tempos de espera correspondentes. Nos intervalos hachurados, o $FPGA_j$ poderia ser utilizado para prover tolerância, durante Δw Unidades de Tempo (UT), para outra(s) tarefa(s) do sistema, "cobrindo" alguns de seus PDFs. Uma reconfiguração poderia ainda ser realizada para retornar a tarefa T_{ij} ao $FPGA_j$ cobrindo o segundo PDF de T_i. Assim como o sistema, as reconfigurações dos FPGAs são periódicas, bastando definir suas ocorrências durante um hiperperíodo.

5.3.2 Detecção de Falha e Substituição de Tarefa

Caso seja indicada uma falha em um dos PDFs cobertos no sistema, o FPGA responsável por sua cobertura é acionado para assumir a execução de sua tarefa. A re-execução da tarefa é realizada e todas as suas tarefas dependentes são executadas tão cedo quanto possível, sendo sus-

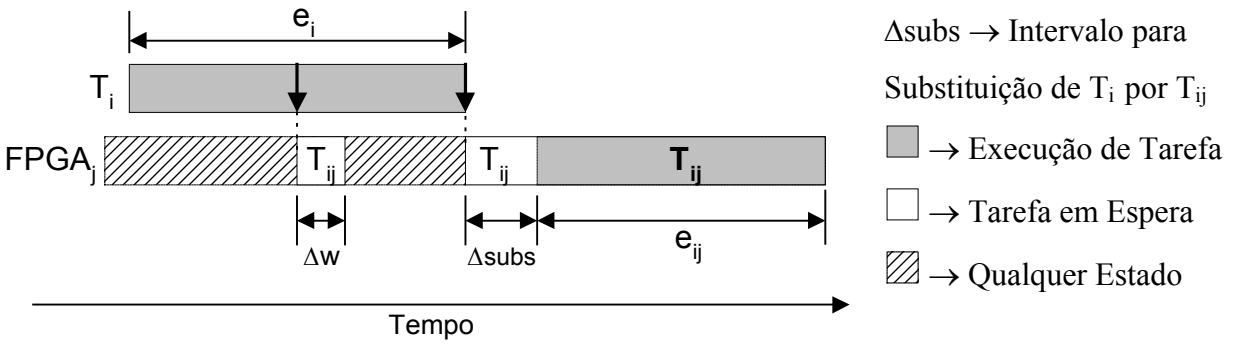


Figura 5.11 Operação do Sistema Durante Falha

penso o escalonamento corrente durante o hiperperíodo do sistema no qual a falha acontece. Essa medida visa a diminuição dos atrasos impostos ao sistema pela re-execução da tarefa falha, além de permitir uma avaliação precisa desses atrasos, como veremos no capítulo seguinte. Na Figura 5.11 é ilustrada a situação em que é indicada uma falha no segundo PDF da tarefa T_i, coberto por $FPGA_j$. Durante um intervalo de tempo Δsubs é realizada a substituição do PE falho, através de chaveamentos dos barramentos ilustrados na parte superior da Figura 5.7. Além disso, são lidos pelo FPGA os dados armazenados no seu *buffer* de entrada (Figura 5.9).

Durante todo o hiperperíodo no qual a falha ocorre, as reconfigurações do $FPGA_j$ são suspensas e ele executa todas as cópias de T_i. No hiperperíodo seguinte ao da falha, o sistema volta a operar num escalonamento, pré-definido, no qual a tarefa falha é escalonada diretamente num FPGA.

5.3.3 Funcionamento após Tratamento de Falha

Passado o hiperperíodo no qual uma tarefa falha, é ativado um escalonamento para o sistema no qual suas execuções são realizadas num FPGA. A determinação de escalonamentos estáticos nos quais tarefas são executadas em FPGAs tem por objetivo restaurar o funcionamento do sistema considerando seus tempos de execução nesses dispositivos. Isso evita que sejam reservados intervalos de tempo maiores (ou menores) que os necessários para a execução das tarefas ao longo do hiperperíodo do sistema. Esses escalonamentos devem estar disponíveis no banco de configurações e escalonamentos do sistema, apresentado na Figura 5.7.

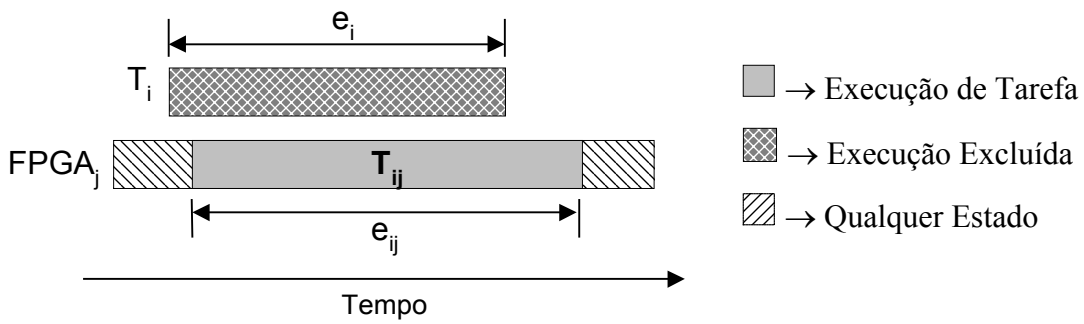


Figura 5.12 Operação do Sistema Após Tratamento de Falha

Na Figura 5.12 é ilustrada a relação entre dois escalonamentos para uma mesma tarefa: a execução no escalonamento sem falhas (T_i) e no escalonamento após o tratamento da falha ocorrida (T_{ij}), com sua execução no $FPGA_j$. A execução no FPGA (e_{ij}) é mostrada deslocada da realizada anteriormente no PE (e_i) para evidenciar que isso pode ocorrer, tanto para a direita quanto para a esquerda em relação ao intervalo de execução sem falha. O intervalo e_{ij} é apresentado maior que o intervalo e_i, o que é comum na prática.

5.4 Síntese de Sistemas Embutidos Empregando Redundância Coletiva

Pela seção anterior, percebe-se a necessidade de *um* escalonamento para a situação de operação normal do sistema e *n* escalonamentos para as situações posteriores ao tratamento de falhas, onde *n* é o número de tarefas tolerantes a falhas do sistema. Na operação normal, todos os PDFs das tarefas tolerantes a falhas devem estar cobertos pela alocação de FPGAs disponível. Em cada escalonamento pós-tratamento de falhas, a tarefa no PE falho deve executar em um dos FPGAs alocados.

Os escalonamentos dos FPGAs incluem as reconfigurações ao longo do hiperperíodo do sistema embutido. Nessas operações são configuradas funcionalidades previamente sintetizadas em *bitstreams* e armazenadas no banco de configurações do sistema (Figura 5.7). Para FPGAs sem reconfiguração parcial, qualquer *bitstream* apresenta a mesma quantidade de bits, não importando quantas funcionalidades são configuradas através dele [2]. Isso implica num tempo

de configuração constante para os dispositivos, em qualquer operação, desde que mantida a mesma freqüência de carregamento dos *bitstreams*. Dessa característica pode-se concluir que, quanto maior o número de funcionalidades configuradas por reconfiguração, melhor aproveitamento será feito do intervalo de tempo ocupado por ela no hiperperíodo do sistema. Isso reduz o número de configurações necessárias para cobrir os PDFs das tarefas com necessidade de tolerância a falhas, reduzindo conseqüentemente o consumo de potência dos FPGAs. Em [53], é citado que operações de reconfiguração dinâmica podem representar até 50% do consumo total de um FPGA.

Com o objetivo de auxiliar no projeto de sistemas embutidos tolerantes a falhas empregando redundância coletiva, foi desenvolvido um algoritmo de síntese capaz de: 1) fornecer uma alocação mínima de FPGAs para funcionar como redundância coletiva para um dado sistema; 2) escalonar um número mínimo de reconfigurações dos FPGAs capaz de cobrir os PDFs das tarefas em hardware dedicado do sistema; 3) escalonar a execução de todas as tarefas do sistema dentro de seus prazos.

A determinação da alocação mínima de FPGAs pode ser realizada priorizando menor custo ou menor área de ocupação. A escolha entre esses dois critérios deve ser realizada pelo projetista antes da execução do algoritmo.

A fim de obter um número mínimo de reconfigurações dos FPGAs, os escalonamentos dos FPGAs são obtidos ao mesmo tempo em que o escalonamento das execuções das tarefas do sistema. Isso permite que as margens de tempo das tarefas sejam exploradas de modo que a minimização do número de reconfigurações seja eficientemente realizada. O conceito de margens de tempo é apresentado no próximo capítulo.

O algoritmo desenvolvido foi denominado RECASTER (***RE****configurable Logi****C AS*** *Fault* ***T****olerac****E*** ***R****esource*), de forma a evidenciar seus objetivos a partir do significado da palavra inglesa *recast*[4]. Sua estrutura é descrita em detalhes no próximo capítulo, incluindo a motivação para algumas das decisões tomadas durante seu desenvolvimento.

[4] **re.cast** [ri:k´a:st;ri:k´æst] *n* **1** refundição. **2** reforma, remodelação. · *vt* **1** refundir. **2** reformar, remodular. **3** tornar a mudar. **4** lançar novamente. **5** redistribuir. **6** tornar a computar.

Resumo do Capítulo 5

Neste capítulo foi introduzida a motivação para o emprego de FPGAs como redundância coletiva, proposta uma arquitetura genérica para a sua realização e descrito seu funcionamento.

A utilização de FPGAs como elementos redundantes permite que diversas tarefas em hardware possam compartilhar um mesmo dispositivo reserva, simultaneamente ou não. Isso possibilita a diminuição da área de hardware do sistema, por não serem necessários dispositivos reserva dedicados para tarefas que necessitem de tolerância a falhas.

A arquitetura proposta para o sistema evidencia a necessidade de um dispositivo responsável pelo gerenciamento do escalonamento do sistema, realizado com base num banco de dados construído durante sua síntese.

O funcionamento do sistema foi descrito com base em quatro hipóteses: 1) tarefas tolerantes a falhas executam em PEs exclusivos; 2) PEs indicam e detectam suas falhas; 3) tarefas falhas são re-executadas integralmente em FPGAs; 4) apenas uma tarefa pode falhar durante um hiperperíodo do sistema. A primeira hipótese condiz com a natureza dedicada e, conseqüentemente exclusiva, de PEs executando tarefas em hardware. Essa característica, aliada a quarta hipótese, permite que o tratamento de falhas consista na substituição de no máximo um PE por hiperperíodo, tornando-o mais simples. A segunda hipótese possibilita um nível de abstração mais elevado com relação ao sistema, tratando conjuntos função + teste como tarefas. A terceira hipótese facilita o tratamento de falhas por não serem considerados resultados parciais internos das tarefas.

Foram consideradas três situações de funcionamento para o sistema: 1) normal, na qual todos os FPGAs funcionam como redundância coletiva em espera, cobrindo PDFs; 2) detecção e substituição de tarefa, que compreende o hiperperíodo no qual um PDF coberto indica uma falha e sua tarefa é executada num FPGA; 3) pós-tratamento de falha, situação na qual o sistema passa à um escalonamento previamente definido, com a tarefa falha executada permanentemente num FPGA. A situação normal é associada ao escalonamento inicial do sistema, enquanto a pós-tratamento de falhas consiste em diversos escalonamentos, associados a cada situação de falha explorada pelo algoritmo de síntese descrito no próximo capítulo.

Capítulo 6

Algoritmo RECASTER

Este capítulo apresenta o algoritmo RECASTER, uma ferramenta computacional desenvolvida para auxiliar no projeto de sistemas embutidos tolerantes a falhas empregando redundância coletiva.

6.1 Introdução

O objetivo principal do algoritmo RECASTER é a síntese de uma estrutura de redundância coletiva para tornar um sistema embutido tolerante a falhas. Denominada neste trabalho de "síntese de redundância", ela consiste na alocação de FPGAs e no mapeamento e escalonamento de tarefas redundantes do sistema nesses dispositivos. O RECASTER realiza essa síntese junto com o escalonamento das tarefas do sistema, com o objetivo de escalonar suas execuções de modo que seus PDFs sejam cobertos empregando um número mínimo de FPGAs e de reconfigurações. Para tanto, se necessário, as execuções das tarefas são atrasadas, desde que isso não implique na violação de prazos especificados.

A Figura 6.1 situa o RECASTER dentro do fluxo geral de síntese de sistema. Como entrada ele deve receber um sistema embutido com alocação de recursos e mapeamento de tarefas pronto e fornecer como saída um escalonamento e uma estrutura de redundância coletiva. Uma vez que a alocação e o mapeamento devem estar completos, a inserção do algoritmo num ambiente de síntese iterativa (ou evolutiva) pode ser realizada pela substituição de sua etapa de escalonamento pelo RECASTER, após realizadas adaptações de formatos de dados.

As próximas seções deste capítulo são dedicadas à descrição do algoritmo desenvolvido e à apresentação de sua aplicação num exemplo ilustrativo.

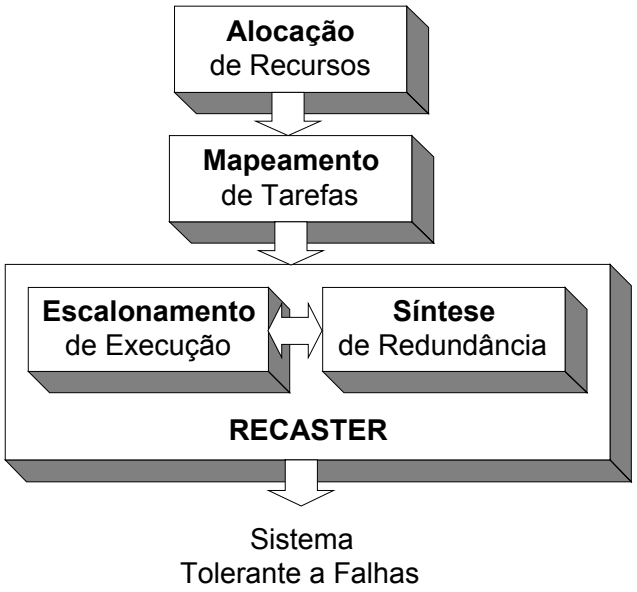


Figura 6.1 RECASTER na Síntese de Sistema Embutido Tolerante a Falhas

6.2 Etapas do Algoritmo RECASTER

O algoritmo RECASTER é composto por três fases: 1- Leitura de Dados e Preenchimento de Estruturas Internas; 2- Escalonamento do Sistema e Síntese de Redundância Coletiva; 3- Apresentação dos Resultados. Na Figura 6.2 é mostrado seu fluxograma geral, com as fases demarcadas por blocos tracejados. Todos os dados necessários para a execução do algoritmo são lidos no início, através de arquivos de entrada. Após essa leitura, diversas estruturas de dados são geradas internamente na etapa de pré-processamento. A segunda fase, a mais importante, é composta por dois *loops*. O mais externo trata da definição da alocação de FPGAs, a ser empregada como redundância coletiva do sistema. O segundo trata da definição do cenário de falha para o qual será obtido um escalonamento das tarefas, as quais podem ou não executar em FPGAs, daí o fato de também serem escalonadas nessa fase reconfigurações desses dispositivos.

Um **cenário** de falha é definido pelas tarefas que se tornam inoperantes e devem ser executadas em FPGAs após a ocorrência de uma falha. Um escalonamento válido para o cenário associado à tarefa T_i, por exemplo, é um escalonamento no qual sua funcionalidade é executada num $FPGA_j$, através da tarefa redundante T_{ij}. Toda falha a ser tolerada precisa ter associado um

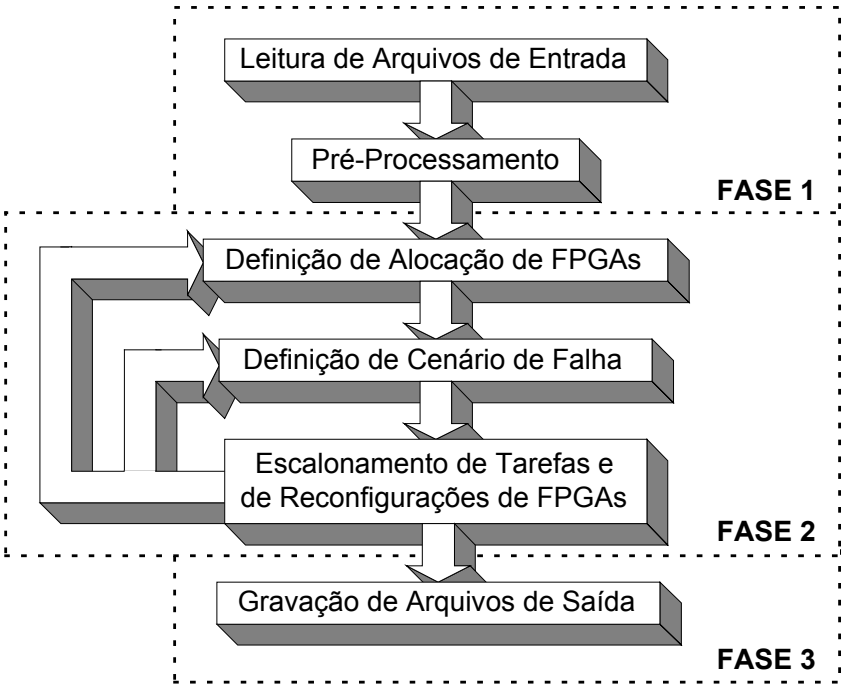


Figura 6.2 Fluxograma Global do Algoritmo RECASTER

escalonamento referente ao funcionamento do sistema após seu tratamento, que opere como descrito na seção 5.3.3 deste trabalho. O escalonamento associado ao funcionamento normal do sistema (seção 5.3.1) é o cenário de ausência de falha, no qual não há tarefas redundantes executando em FPGAs. Nesse caso, esses dispositivos são reconfigurados periodicamente e apenas mantidos como reservas.

Após a obtenção de escalonamentos válidos para todos os cenários de falha especificados, empregando a alocação de FPGAs definida no *loop* mais externo, o algoritmo passa a fase 3, no qual os armazena em arquivos. Nessa fase também é gerado um arquivo contendo os seguintes parâmetros, empregados na avaliação da solução obtida: custo e área total da alocação de FPGAs empregada como redundância coletiva; violação percentual máxima e média de prazos em caso de falha e re-execução das tarefas do sistema.

A seguir as fases do algoritmo são apresentadas detalhadamente. Cada fase é associada à um fluxograma e subdividida em etapas numeradas para auxiliar na sua descrição.

6.2.1 Fase 1: Leitura de Dados e Preenchimento de Estruturas Internas

A Figura 6.3 apresenta o fluxograma da fase 1 do algoritmo. Os arquivos lidos durante as duas primeiras etapas são responsáveis pela construção da biblioteca de FPGAs e das listas de tarefas que especificam o sistema. Os FPGAs da biblioteca são utilizados na formação de alocações candidatas à redundância coletiva, enquanto as listas de tarefas contêm as mesmas informações dos grafos da especificação funcional do sistema e também o mapeamento de suas tarefas numa alocação.

Biblioteca de FPGAs (Etapa 1)

A biblioteca de FPGAs consiste numa lista de dispositivos, cada um com as seguintes informações disponíveis:

- *área* do encapsulamento (cm^2) → empregado na ordenação e avaliação das alocações;
- *custo* (US$) → empregado na ordenação e avaliação das alocações;
- *capacidade lógica* (Unidades Lógicas-UL) → quantidade total de recursos lógicos programáveis do dispositivo, em sua menor unidade componente, excluído o nível de portas[5];
- *# de pinos programáveis* (pinos) → quantidade de pinos disponíveis para configurações;
- *# de bits de configuração* (Kbits) → quantidade de bits necessários para realizar a programação completa do dispositivo;
- *freqüência máxima de reconfiguração* (MHz) → máxima freqüência para reconfiguração por carregamento serial;
- Δw (UT) → tempo necessário para ação de tratamento de falhas, associado diretamente ao FPGA para simplificação do modelo de entrada do algoritmo;
- *quantidade* (unidades) → número máximo de unidades de um FPGA;
- *lista de tarefas* que têm uma versão executável em FPGA, apresentando para cada versão:
  - *taxa de ocupação* (%) → razão entre o número de unidades lógicas que a versão requer e a capacidade lógica do FPGA (x 100%);
  - *tempo de execução* (UT) → quantidade de tempo gasto na sua execução.

[5] o número de portas lógicas equivalentes fornecido por fabricantes corresponde à, aproximadamente, sua capacidade lógica em portas NAND de 2 entradas [33, 10].

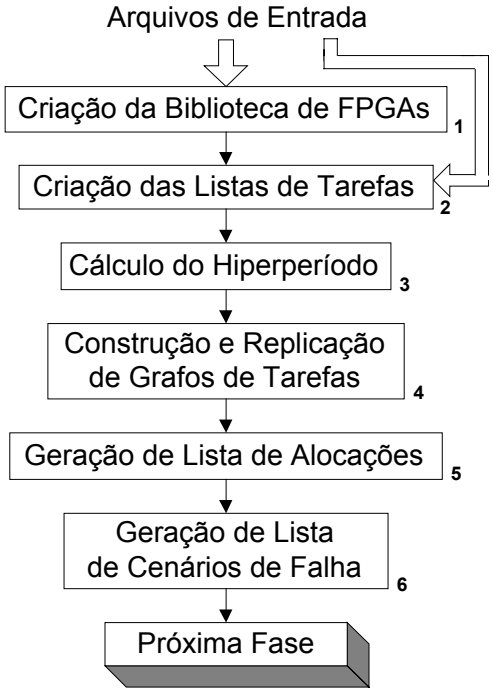


Figura 6.3 Fluxograma da Fase 1

Listas de Tarefas (Etapa 2)

Para cada grafo de especificação do sistema embutido é criada uma lista de tarefas, associada ao seu período de execução. Cada tarefa apresenta:

- *prazo* (UT);
- *EST* (UT);
- *lista de PDFs* com relação ao seu tempo de execução (%);
- indicador de *necessidade de redundância* (Sim/Não) → preenchido pelo projetista;
- *PE*, com os seguintes valores associados:
  - *área* ocupada por encapsulamento (cm^2);
  - *tempo de execução* da tarefa (UT).
- *# de pinos de interface* → total de pinos envolvidos na realização da tarefa, que devem existir tanto no seu PE quanto num FPGA que o substitua;
- *listas de arcos* de entrada e de saída, cada um apresentando:
  - *link*, com:
    - *tempo de comunicação* (UT) associado.

Cálculo do Hiperperíodo (Etapa 3)

O hiperperíodo é calculado a partir dos períodos dos grafos de tarefas usados para especificar o sistema. Como dito anteriormente, esse valor corresponde ao mínimo múltiplo comum (*mmc*) dos períodos. Para quaisquer dois inteiros positivos a e b, é válida a relação:

$$mmc(\mathrm{a},\mathrm{b}) = \frac{|\mathrm{ab}|}{mdc(\mathrm{a},\mathrm{b})} \text{ [48]}, \tag{6.1}$$

que permite obter o *mmc* dos números a partir do máximo divisor comum (*mdc*) dos mesmos. O RECASTER emprega essa relação e utiliza um procedimento de cálculo de mdc denominado *processo das divisões sucessivas* [48], bastante conhecido e de fácil codificação nas principais linguagens de programação existentes.

Quando o número de grafos é maior que dois, a equação 6.1 também pode ser aplicada, já que, para três inteiros positivos a, b e c, $mmc(\mathrm{a},\mathrm{b},\mathrm{c}) = mmc(mmc(\mathrm{a},\mathrm{b}),\mathrm{c})$, relação que pode ser aplicada na determinação do *mmc* de qualquer conjunto de números inteiros.

Construção e Replicação de Grafos de Tarefas (Etapa 4)

Nesta etapa são construídos grafos com base nas informações dos arcos de entrada e saída das listas de tarefas. Cada grafo é replicado de acordo com o número de execuções durante o hiperperíodo do sistema. Esse número corresponde a razão entre o hiperperíodo do sistema e o período do grafo.

O EST e o prazo da execução de número Nc da tarefa T_i (T_i^{Nc}) do grafo de período P são dados por:

$$\mathrm{Prazo}_i^{Nc} = \mathrm{Prazo}_i + \mathrm{Nc} \cdot \mathrm{P} \tag{6.2}$$

$$\mathrm{EST}_i^{Nc} = \mathrm{EST}_i + \mathrm{Nc} \cdot \mathrm{P} \tag{6.3}$$

Geração de Lista de Alocações (Etapa 5)

Nessa etapa são formadas alocações de FPGAs para atuarem como redundância coletiva. Para cada FPGA na biblioteca são formadas alocações homogêneas contendo de uma à *X* unidades do mesmo, sendo *X* igual a sua *quantidade*. Cada alocação tem os seguintes parâmetros:

- *tipo* → tipo de FPGA utilizado na alocação;
- *# de unidades* (unidades) → quantidade de dispositivos utilizados na alocação;
- *área* (cm^2) → produto da *área* do FPGA empregado por *# de unidades* na alocação;
- *custo* (US$) → produto do *custo* do FPGA empregado por *# de unidades* na alocação;
- *listas de unidades* → dados relativos ao escalonamento individual de cada dispositivo.

Após formadas todas as alocações, é gerada uma lista na qual são dispostas em ordem crescente de *custos* ou de *áreas*, dependendo do critério de otimização escolhido pelo projetista. A figura 6.4 mostra em pseudocódigo o procedimento realizado na geração dessa lista.

Geração de Lista de Cenários de Falhas (Etapa 6)

Cada cenário de falha é especificado pelo seu conjunto de tarefas afetadas. O algoritmo considera apenas falhas simples, isto é, somente uma tarefa afetada por cenário. Assim, para *n* tarefas que necessitam de redundância, são gerados *n*+1 cenários, sendo o cenário adicional correspondente à ausência de falhas (∅). A síntese de redundância coletiva deve manter as tarefas do sistema disponíveis para todos esses cenários.

```
GERA_LISTA_DE_ALOCAÇÕES (Biblioteca_de_FPGAs, Critério_de_Otimização)
    α=1;   // Índice para Alocações
    para cada FPGA_j da Biblioteca
        para (k=1 até k=quantidade_j)
            tipo_α= FPGA_j;
            custo_α= custo_j · k;
            área_α= área_j · k;
            #_de_unidades_α= k;
            α=α+1;
    se (Critério_de_Ordenação == Custo)
        Ordena Alocações na Ordem Crescente dos Custos;
    senão
        Ordena Alocações na Ordem Crescente das Áreas;
```

Figura 6.4 Procedimento de Geração da Lista de Alocações

6.2.2 Fase 2: Escalonamento e Síntese de Redundância Coletiva

A Figura 6.5 apresenta o fluxograma da fase 2 do algoritmo e na seqüência são descritas suas principais etapas.

Escolha da Próxima Alocação (Etapa 1)

Essa etapa é responsável pela escolha da alocação a ser explorada na síntese de redundância coletiva para o sistema. Denominada *alocação corrente*, na primeira execução ela corresponde a primeira da lista de alocações gerada na fase 1, sendo portanto a de menor área ou custo, conforme o critério de otimização adotado.

Sempre que executada, essa etapa é responsável por definir a *alocação corrente* como a próxima da lista. Quando essa alocação não for suficiente para a síntese de redundância do sistema, essa etapa será executada novamente para determinar a próxima alocação a ser explorada.

Escolha do Próximo Cenário de Falhas (Etapa 2)

A definição da *alocação corrente* é seguida pela definição do cenário de falhas para o qual será obtido um escalonamento do sistema. Essa etapa é repetida para cada cenário da lista gerada na fase 1.

Determinação das Margens de Tempo Iniciais das Tarefas (Etapa 3)

Os valores iniciais das margens de tempo das tarefas são calculados com base no tipo de FPGA da *alocação corrente* e no cenário de falhas escolhido na etapa anterior. Entende-se por margem de tempo o máximo atraso (em UT) no início de execução de uma tarefa sem que sejam ultrapassados (ou violados) os prazos do sistema.

O procedimento de escalonamento realizado pelo RECASTER é baseado em lista de tarefas prontas. Esse método consiste em se escalonar tarefas retiradas individualmente do topo

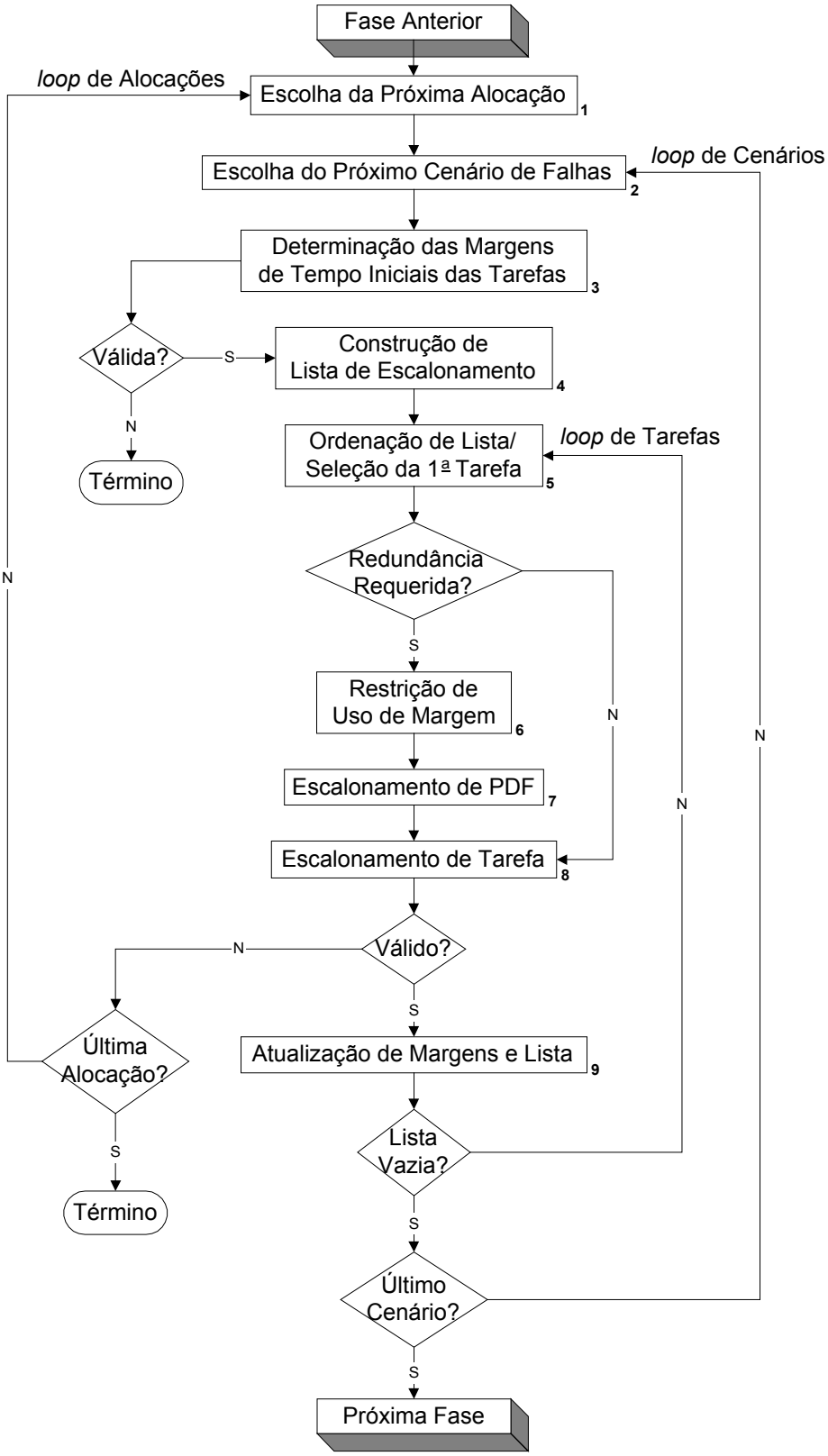


Figura 6.5 Fluxograma da Fase 2

de uma lista de tarefas, não escalonadas, cujas dependências de comunicação/controle já foram satisfeitas. Na lista, essas tarefas são ordenadas por alguma métrica, denominada prioridade, uma das características mais relevantes na diferenciação entre os diversos algoritmos de escalonamento baseados em lista existentes. Entre as métricas mais comuns pode-se citar os inícios de execução ASAP (*As Soon As Possible*) e ALAP (*As Late As Possible*) das tarefas, suas mobilidades [23] ou folgas [15]. Os inícios ASAP e ALAP de uma tarefa são, respectivamente, os instantes mais cedo e mais tarde possíveis de se iniciar sua execução, assumindo que existem recursos de processamento ilimitados. A folga de uma tarefa é a diferença entre esses dois tempos. A mobilidade é análoga à folga, não sendo porém medida em unidade de tempo, e sim em passos de controle ou estágios de execução de um sistema.

O problema geralmente associado às métricas anteriores é o fato de serem estáticas, impedindo que mudanças nos intervalos de execução de tarefas, durante seus escalonamentos, reflitam na prioridade daquelas que ainda serão escalonadas. Prioridades dinâmicas têm sido sugeridas para evitar isso [47, 53].

A métrica empregada no RECASTER é dinâmica e se baseia no intervalo de execução mais cedo possível de uma tarefa. A determinação de sua margem de tempo dependente diretamente dessa métrica.

A margem de tempo total de uma tarefa T_i é:

$$M\tau_{total}(T_i) = Lst(T_i) - Est(T_i), \tag{6.4}$$

$Lst(T_i) \rightarrow$ início mais tarde de execução de T_i, com restrição de recursos de processamento;

$Est(T_i) \rightarrow$ início mais cedo de execução de T_i, com restrição de recursos de processamento.

Esses tempos dependem do mapeamento das tarefas nos recursos de processamento (PEs ou FPGAs) do sistema, que definem o tempo que gastam para executar, além das suas dependências de comunicação/controle. Caso cada tarefa tenha sido mapeada em um processador exclusivo, a obtenção dos seus *Lst* e *Est* reduz-se a determinação de seus inícios ALAP e ASAP. No entanto, o RECASTER não impõe tal restrição aos sistemas de entrada, pois isso impediria trabalhar-se com sistemas hardware/software com compartilhamento de PEs.

Múltiplas tarefas mapeadas num mesmo PE devem ter suas execuções ordenadas de modo que seus *Lst* e *Est* possam ser determinados. Assim, foi desenvolvido um método para realizar essa ordenação baseada nos escalonamentos ASAP e ALAP das tarefas, para só após isso serem

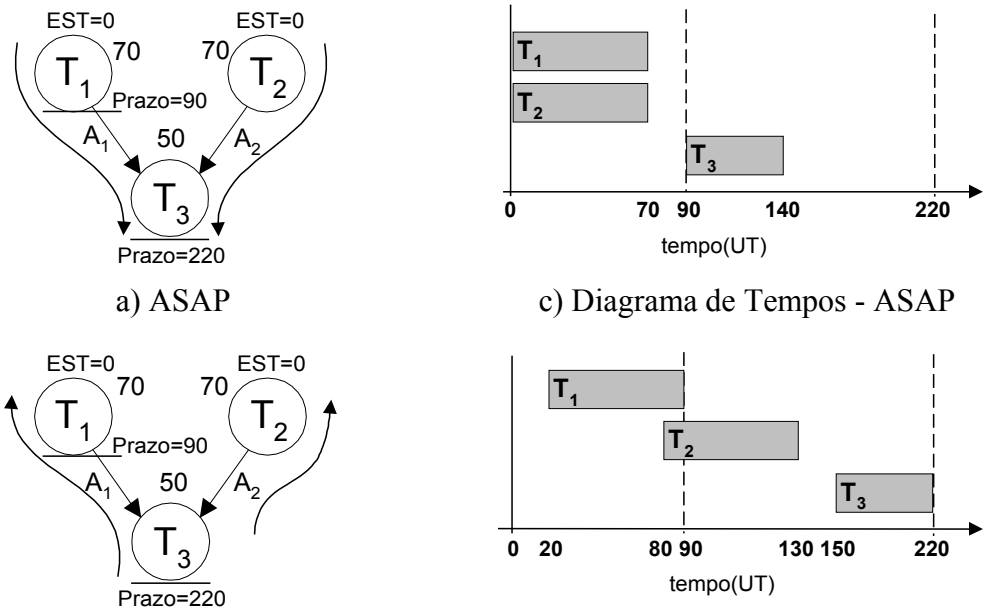
obtidos seus *Lst*, *Est* e margens. A seguir são descritos os procedimentos de determinação dos escalonamentos ASAP e ALAP das tarefas e dos seus *LATE* e *EARLY*. Esses últimos são equivalentes aos ASAP e ALAP, porém com restrição de recursos de processamento e ordenação de execução de tarefas seguindo a estratégia definida neste trabalho. O valor *Lst* (*Est)* corresponde ao instante inicial de execução de um tarefa no seu escalonamento *LATE (EARLY).*

ASAP e ALAP

Os escalonamentos ASAP e ALAP de um sistema são obtidos através de buscas topológicas ao longo dos seus grafos de especificação. Em [8], esses procedimentos são apresentados através de equações. Aqui, no entanto, serão mostrados e exemplificados através de pseudocódigos dos procedimentos utilizados no RECASTER.

Nas Figuras 6.6a-b as setas curvas apresentam o sentido das buscas topológicas realizadas num grafo exemplo. Os tempos de execução, ESTs e prazos das tarefas são apresentados junto aos nós. O período do grafo é 220 e os tempos de comunicação 20 em todos os arcos. Todos os valores correspondem a UT.

A Figura 6.7 apresenta em pseudocódigo o procedimento adotado para a obtenção do escalonamento ASAP de um grafo de tarefas. Aplicando-o ao grafo da Figura 6.6, a execução da



a) ASAP

c) Diagrama de Tempos - ASAP

b) ALAP

d) Diagrama de Tempos – ALAP

Figura 6.6 Escalonamentos ASAP e ALAP

tarefa de entrada T_1 é atribuída ao intervalo de tempo (0,70), por apresentar EST=0 e tempo de execução (*Tempo_de_Execução*$_1$) igual a 70. Uma vez que seu prazo não é violado e ela possui um arco de saída, o procedimento auxiliar ASAP_FILHAS é chamado com os argumentos T_3 e 90, este último correspondendo ao menor início de execução possível para T_3. Em ASAP_FILHAS, é assegurado que T_3 inicie no maior instante entre todos os que lhe foram atribuídos como menor início possível. Isso garante que sua execução ocorra após satisfeitas suas dependências de comunicação. Pelo fato de T_3 não ter sido visitada anteriormente, *ASAP*$_3$*.Início* é zero, e sua execução é atribuída ao intervalo (90,90+*Tempo_de_Execução*$_3$) = (90,140). Após isso, as condições para que o procedimento continue sem violações de EST e prazo são testadas e satisfeitas. Por T_3 não possuir arcos de saída, retorna-se ao procedimento principal, especificamente para o primeiro *loop* **para**, já que T_1 não possui outro arco de saída além de A_1. Tomando-se agora a outra tarefa de entrada, T_2, todo o procedimento anterior é repetido. Sua execução é atribuída ao intervalo (0,70) e a de T_3, novamente visitada, permanece em (90,140).

A Figura 6.8 apresenta em pseudocódigo o procedimento empregado na obtenção do escalonamento ALAP de um grafo de tarefas. Com o grafo da Figura 6.6 como entrada, a única

```
ASAP (Grafos_de_Tarefas)
    para cada Ti de Entrada
        ASAPi.Início= ESTi;
        ASAPi.Término= ESTi+Tempo_de_Execuçãoi;
        se (ASAPi.Término > Prazoi)
            Interrompe e Sinaliza Falha no Escalonamento de Ti ;
        se (Arcos_de_Saídai ≠ ∅)
            para cada Arco de Saída (An) de Ti
                ASAP_FILHAS(Tarefa Apontada por An, ASAPi.Término+Tempo_de_Comunicaçãon);
```

```
ASAP_FILHAS (Tβ, Menor_Início)
    se (Menor_Início > ASAPβ.Início)
        ASAPβ.Início= Menor_Início;
        se (ASAPβ.Início < ESTβ)
            ASAPβ.Início= ESTβ;
        ASAPβ.Término= ASAPβ.Início + Tempo_de_Execuçãoβ;
    se (ASAPβ.Início < ESTβ) ou  (ASAPβ.Término > Prazoβ)
        Interrompe e Sinaliza Falha no Escalonamento de Tβ;
    se (Arcos_de_Saídaβ ≠ ∅)
        para cada Arco de Saída (An) de Tβ
            ASAP_FILHAS(Tarefa Apontada por An, ASAPβ.Término+Tempo_de_Comunicaçãon);
```

Figura 6.7 Procedimento de Escalonamento ASAP

tarefa de saída, T_3, é atribuída ao intervalo de tempo (220-*Tempo_de_Execução*$_3$,220) = (170, 220). O limite superior corresponde ao seu prazo, e o inferior a diferença (prazo- *Tempo _de_Execução*$_3$). Uma vez que EST_3 não é violado e T_3 possui dois arcos de entrada, o procedimento auxiliar ALAP_PAIS é chamado duas vezes. Na primeira, os argumentos são T_1 e 150, este último correspondendo ao maior término de execução possível para T_1. Em ALAP_PAIS, é assegurado que a tarefa no argumento termine no menor instante entre todos os que lhe foram atribuídos como maior término possível, o que garante que sua execução não implicará na violação de prazos de suas dependentes. Pelo fato de T_1 não ter sido visitada anteriormente, $ALAP_1.Término$ é ∞. O valor 150 é atribuído à $ALAP_1.Término$ mas não permanece nele por ser maior que $Prazo_1$. A execução de T_1 é então atribuída ao intervalo (90-*Tempo_de_Execução*$_1$,90) = (20,90). Como não há violações de prazo nem de EST, a execução do procedimento continua. Pelo fato de T_1 não possuir arcos de entrada, retorna-se ao procedimento principal e ALAP_PAIS é chamado com argumentos os T_2 e 150. Por fim, T_2 é atribuída ao intervalo de execução (80,150). As Figuras 6.6c-d apresentam os diagramas de tempos dos escalonamentos ASAP e ALAP obtidos.

```
ALAP (Grafos_de_Tarefas)
    para Cada Tarefa Ti de Saída
        ALAPi.Término= Prazoi;
        ALAPi.Início= Prazoi - Tempo_de_Execuçãoi;
        se (ALAPi.Início < ESTi)
            Interrompe e Sinaliza Falha no Escalonamento de Ti;
        se (Arcos_de_Entradai ≠ ∅)
            para cada Arco de Entrada (Am) de Ti
                ALAP_PAIS (Tarefa que Aponta Am, ALAPi.Início -Tempo_de_Comunicaçãom);
```

```
ALAP_PAIS (Tβ, Maior_Término)
    se (Maior_Término < ALAPβ.Término)
        ALAPβ.Término= Maior_Término;
        se (ALAPβ.Término > Prazoβ)
            ALAPβ.Término=Prazoβ;
        ALAPβ.Início= ALAPβ.Término - Tempo_de_Execuçãoβ;
    se (ALAPβ.Término > Prazoβ) ou (ALAPβ.Início < ESTβ)
        Interrompe e Sinaliza Falha no Escalonamento de Tβ;
    se (Arcos_de_Entradaβ ≠ ∅)
        para cada Arco de Entrada (Am) de Tβ
            ALAP_PAIS (Tarefa que Aponta Am, ALAPβ.Início-Tempo_de_Comunicaçãom);
```

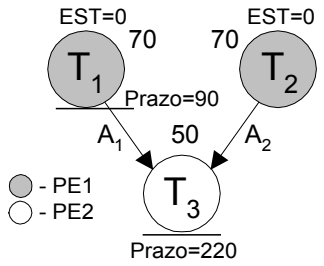
Figura 6.8 Procedimento de Escalonamento ALAP

EARLY e LATE

A Figura 6.9a repete o grafo da Figura 6.6, agora evidenciando o compartilhamento do PE1 entre T_1 e T_2. Tarefas que compartilham um mesmo PE têm suas execuções ordenadas segundo seus instantes finais no escalonamento ALAP. Tarefas com instantes finais menores têm maior prioridade para executar em seus PEs. A partir disso, é criada uma lista de PEs, na qual cada um deles tem suas tarefas ordenadas segundo suas prioridades de execução (Figura 6.9b).

O escalonamento *EARLY* é obtido atribuindo-se a execução das tarefas de cada PE aos seus intervalos mais cedo possíveis, a partir daquelas com maior prioridade. Esses intervalos podem ser diretamente seus escalonamentos ASAP, caso não se sobreponham à execução de tarefas com maior prioridade. Desse modo, antes de chamar o procedimento EARLY, é feito em todas as tarefas do sistema: *EARLY.Início=ASAP.Início; EARLY.Término= ASAP.Término*.

A Figura 6.10 apresenta em pseudocódigo o procedimento para a obtenção do escalonamento EARLY do sistema. Em cada PE são comparados, sucessivamente, os intervalos de execução EARLY de suas tarefas. Sendo detectada uma sobreposição, a tarefa de menor prioridade é atrasada para iniciar imediatamente após o término da anterior. Se esse atraso resultar na violação do prazo da tarefa, o procedimento é interrompido. Caso contrário, suas dependentes têm seus intervalos de execução EARLY atualizados pelo procedimento EARLY_FILHAS. Nele, se necessária, é feita tal atualização e verificada uma possível violação de prazo da tarefa chamada. Havendo violação, o procedimento de escalonamento é interrompido.



a) Compartilhamento de PEs

ASAP:

$T_1 \rightarrow (0,70)$
$T_2 \rightarrow (0,70)$
$T_3 \rightarrow (90,140)$

ALAP:

$T_1 \rightarrow (20,90)$
$T_2 \rightarrow (80,150)$
$T_3 \rightarrow (170,220)$

-Lista_de _PEs={PE1, PE2};
-Ordenação por Instante Final ALAP:
PE1={T_1, T_2};
PE2={T_3};

b) Ordenação de Tarefas

Figura 6.9 Lista de PEs e Ordenação de Tarefas

```
EARLY (Grafos_de_Tarefas, Lista_de_PEs)
    para (l=1 até l=(posição do último PE em Lista_de_PEs)
        para (k=1 até k=(posição da penúltima tarefa em Lista_de_PEs[l]))
            TA=Lista_de_PEs[l][k]; TB=Lista_de_PEs[l][k+1];
            se (EARLY_TA.Término > EARLY_TB.Início)
                EARLY_TB.Início = EARLY_TA.Término;
                EARLY_TB.Término = EARLY_TA.Término + Tempo_de_Execução_TB;
                se (EARLY_TB.Término > Prazo_TB)
                    Interrompe e Sinaliza Falha no Escalonamento de TB;
                se (Arcos_de_Saída_TB ≠ ∅)
                    para cada Arco de Saída (A_n) de TB
                        EARLY_FILHAS (Tarefa Apontada por A_n , EARLY_TB.Término+
                        Tempo_de_Comunicação_n, l);
```

```
EARLY_FILHAS (T_β, Menor_Início, l)
    se (Menor_Início > EARLY_β.Início)
        EARLY_β.Início = Menor_Início;
        EARLY_β.Término = Menor_Início +Tempo_de_Execução_β;
        se (EARLY_β.Término > Prazo_β)
            Interrompe e Sinaliza Falha no Escalonamento de T_β;
        se (posição de PE_β em Lista_de_PEs < l)
            para (k=(posição de T_β em PE_β) até k=(posição da penúltima tarefa em PE_β))
                TA=Lista_de_PEs[posição de PE_β][k]; TB=Lista_de_PEs[posição de PE_β][k+1];
                se (EARLY_TA.Término > EARLY_TB.Início)
                    EARLY_TB.Início = EARLY_TA.Término;
                    EARLY_TB.Término = EARLY_TA.Término + Tempo_de_Execução_TB;
                    se (EARLY_TB.Término > Prazo_TB)
                        Interrompe e Sinaliza Falha no Escalonamento de TB;
                    se (Arcos_de_Saída_TB ≠ ∅)
                        para cada Arco de Saída (A_n) de TB
                            EARLY_FILHAS (Tarefa Apontada por A_n , EARLY_TB.Término+
                            Tempo_de_Comunicação_n, l);
        se (Arcos_de_Saída_β ≠ ∅)
            para cada Arco de Saída (A_n) de T_β
                EARLY_FILHAS (Tarefa Apontada por A_n , EARLY_β.Término + Tempo_de_Comunicação_n, l);
```

Figura 6.10 Procedimento de Escalonamento EARLY

O deslocamento do intervalo EARLY da tarefa pode provocar a sobreposição de sua execução com a de outra (de menor prioridade) em seu PE. Caso tal PE ainda não tenha sido considerado no *loop* mais externo do procedimento EARLY, a possível sobreposição será tratada mais adiante, e a atualização dos intervalos EARLY das dependentes da tarefa segue sendo realizada. Caso contrário, os intervalos EARLY das tarefas de menor prioridade são ajustados como no procedimento EARLY, atualizando-se em seguida os intervalos de suas dependentes.

Executando o procedimento EARLY tendo como argumentos de entrada o grafo da Figura 6.9a e a lista de PEs da Figura 6.9b, têm-se TA=T_1 e TB=T_2. Uma vez que $EARLY_1.Término$ =70 é maior que $EARLY_2.Início$=0, a execução EARLY de T_2 é atrasada para ($EARLY_1$.*Término*, $EARLY_1.Término+Tempo\_de\_Execução_2$)=(70,140). Na seqüência, o procedimento EARLY _FILHAS é chamado com os argumentos T_3, 160 e 1. Nele, pelo fato de $EARLY_3.Início$=90 ser menor que 160, o intervalo de execução EARLY de T_3 é atualizado para (160,210). Uma vez que $EARLY_3.Término$=210 é menor que $prazo_{T3}$, o procedimento continua executando. Pelo fato do PE de T_3 (PE2) ainda não ter sido considerado no loop mais externo do procedimento EARLY e de T_3 não possuir arcos de saída, a execução retorna ao corpo desse último. Uma vez que todos os arcos de T_2 já foram chamados em EARLY_FILHAS e todas as tarefas de PE1 escalonadas, PE2 é considerado para o escalonamento de suas tarefas. Por ter apenas uma tarefa, nenhum procedimento de ajuste é necessário, e o intervalo EARLY de T_3 permanece em (160, 210).

O escalonamento *LATE* tem um procedimento de obtenção bastante semelhante ao descrito para o EARLY, e consiste em atribuir-se a execução das tarefas de cada PE aos seus intervalos mais tarde possíveis, a partir daquelas com menor prioridade. Novamente, esses intervalos podem ser diretamente os escalonamentos ALAP das tarefas, desde que não ocorram sobreposições de execução entre aquelas que compartilham um mesmo PE. Assim, antes de iniciar o procedimento LATE, é feito em todas as tarefas do sistema: *LATE.Início=ALAP.Início; LATE.Término= ALAP.Término*.

A Figura 6.11 apresenta em pseudocódigo o procedimento para a obtenção do escalonamento LATE do sistema. Em cada PE são comparados, sucessivamente, os intervalos de execução LATE de suas tarefas, partindo daquelas com menor prioridade. Sendo detectada uma sobreposição, a tarefa de maior prioridade é adiantada para que termine imediatamente antes do início da posterior. Se esse deslocamento resultar na violação do EST da tarefa, o procedimento é interrompido. Caso contrário, suas tarefas de entrada têm os intervalos de execução LATE atualizados pelo procedimento LATE_PAIS. Nele, se necessária, é feita tal atualização e verificada uma possível violação de EST da tarefa chamada. Havendo violação, o procedimento de escalonamento é interrompido. Uma vez que o deslocamento do intervalo LATE da tarefa pode provocar sobreposições, caso seu PE ainda não tenha sido considerado no loop mais externo do procedimento LATE, a atualização dos intervalos LATE das dependentes da tarefa segue

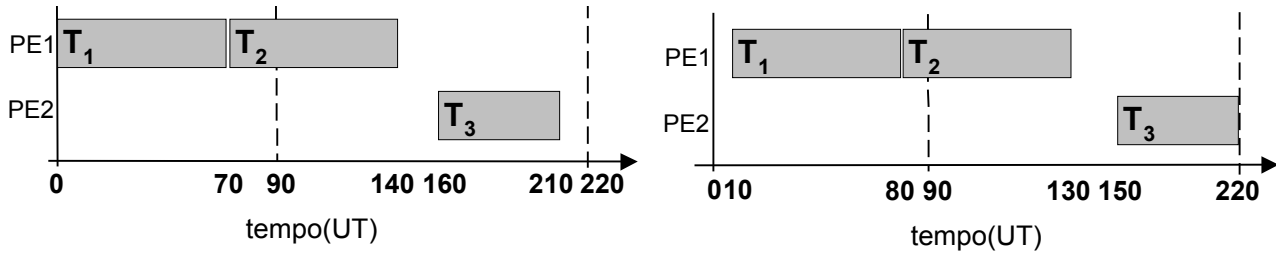
```
LATE (Grafos_de_Tarefas, Lista_de_PEs)
    para (l=1 até l=(posição do último PE em Lista_de_PEs)
        para (k=(Número de Tarefas no PE) até k=2))
            TA=Lista_de_PEs[l][k]; TB=Lista_de_PEs[l][k-1];
            se (LATE_TA.Início < LATE_TB.Término)
                LATE_TB.Término = LATE_TA.Início;
                LATE_TB.Início = LATE_TA.Início - Tempo_de_Execução_TB;
                se (LATE_TB.Início > EST_TB)
                    Interrompe e Sinaliza Falha no Escalonamento de TB;
                se (Arcos_de_Entrada_TB ≠ ∅)
                    para cada Arco de Entrada (A_m) de TB
                        LATE_PAIS (Tarefa Apontada por A_m , LATE_TB.Início-
                        Tempo_de_Comunicação_m, l);
```

```
LATE_PAIS (T_β, Maior_Término, l)
    se (Maior_Término < LATE_β.Término)
        LATE_β.Término = Maior_Término;
        LATE_β.Início = Maior_Término+Tempo_de_Execução_β;
        se (LATE_β.Início < EST_β)
            Interrompe e Sinaliza Falha no Escalonamento de T_β;
        se (posição de PE_β em Lista_de_PEs < l)
            para (k=(posição de T_β em PE_β) até k=2)
                TA=Lista_de_PEs[posição de PE_β][k]; TB=Lista_de_PEs[posição de PE_β][k-1];
                se (LATE_TA.Início < LATE_TB.Término)
                    LATE_TB.Término = LATE_TA.Início;
                    LATE_TB.Início = LATE_TA.Início - Tempo_de_Execução_TB;
                    se (LATE_TB.Início > EST_TB)
                        Interrompe e Sinaliza Falha no Escalonamento de TB;
                    se (Arcos_de_Entrada_TB ≠ ∅)
                        para cada Arco de Entrada (A_m) de TB
                            LATE_PAIS (Tarefa Apontada por A_m , LATE_TB.Início-
                            Tempo_de_Comunicação_m, l);
        se (Arcos_de_Entrada_β ≠ ∅)
            para cada Arco de Entrada (A_m) de T_β
                LATE_PAIS (Tarefa Apontada por A_m , LATE_β.Término - Tempo_de_Comunicação_m, l);
```

Figura 6.11 Procedimento de Escalonamento LATE

sendo realizada. Caso contrário, os intervalos LATE das tarefas de maior prioridade no PE são ajustados e os de suas dependentes atualizados na seqüência. O procedimento LATE para o exemplo da Figura 6.9 resulta nos intervalos LATE $T_1 \rightarrow (10,80)$, $T_2 \rightarrow (80,150)$ e $T_3 \rightarrow (170,220)$. As Figuras 6.12a e 6.12b apresentam, através de diagramas de tempos, os escalona-mentos *EARLY* e *LATE* para todas as tarefas.

As margens de tempo iniciais obtidas para as tarefas do grafo da Figura 6.8a foram todas iguais a 10 UT. O atraso de uma tarefa pode se propagar através de suas dependentes e também



a) Diagrama de Tempos: *EARLY* b) Diagrama de Tempos: *LATE*

Figura 6.12 Escalonamentos *EARLY* e *LATE*

através das tarefas que executam posteriormente em seu PE. Isso torna necessária a atualização de seus intervalos *EARLY* e conseqüentemente de suas margens. Um atraso de 5 UT em T_1 levaria sua execução para o intervalo de tempo (5,75), reduzindo para 5 UT sua margem de tempo. Com T_1 ocupando esse intervalo, T_2 não poderá manter-se em (70,140), devendo ser deslocada para (75,145), reduzindo também sua margem para 5 UT. Finalmente, em decorrência das alterações anteriores, T_3 passa a ser escalonada em (165,215), restando 5 UT de margem. Os intervalos obtidos para T_2 e T_3 são seus novos escalonamentos *EARLY*, e quaisquer atrasos de suas execuções devem considerá-los como base, além de serem limitados pelas novas margens. Percebe-se através desse pequeno exemplo o caráter compartilhado das margens das tarefas.

O procedimento de escalonamento do RECASTER é baseado no atraso seguido da atualização de tempos de execução *EARLY* das tarefas, como será explicado ao longo das seções seguintes. A interrupção de qualquer um dos procedimentos de escalonamento descritos anteriormente (ASAP, ALAP, EARLY e LATE) implica na obtenção de margens iniciais inválidas e, conseqüentemente, no término da execução do algoritmo (Figura 6.5).

Construção de Lista de Tarefas Prontas (Etapa 4)

Nessa etapa é criada uma lista das tarefas prontas para escalonamento, a qual é inicialmente constituída por todas as tarefas de entrada dos grafos de especificação do sistema. Uma vez que a topologia dos grafos permanece inalterada durante a execução do algoritmo, qualquer repetição dessa etapa resultará em listas com as mesmas tarefas.

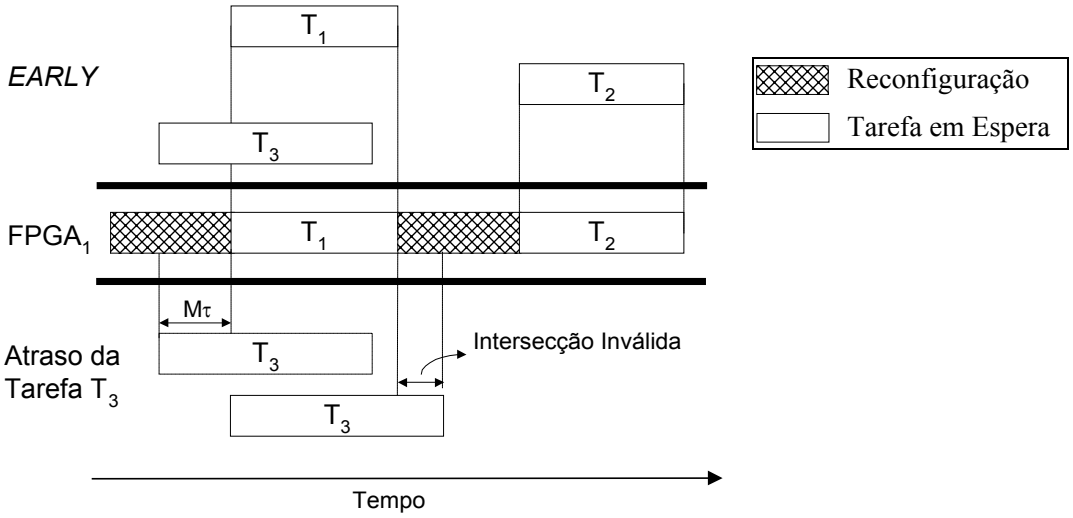
Ordenação de Lista de Tarefas Prontas (Etapa 5)

As tarefas na lista são dispostas na ordem crescente da próxima ocorrência de um PDF não-coberto conforme o escalonamento *EARLY* do sistema, caso necessitem redundância. Para tarefas sem redundância ou implementadas através de software, o parâmetro de ordenação é o instante inicial de execução no seu escalonamento *EARLY*, o que permite que sejam consideradas para escalonamento o mais cedo possível. Por não apresentarem PDFs para serem escalonados, atrasá-las torna-se desnecessário.

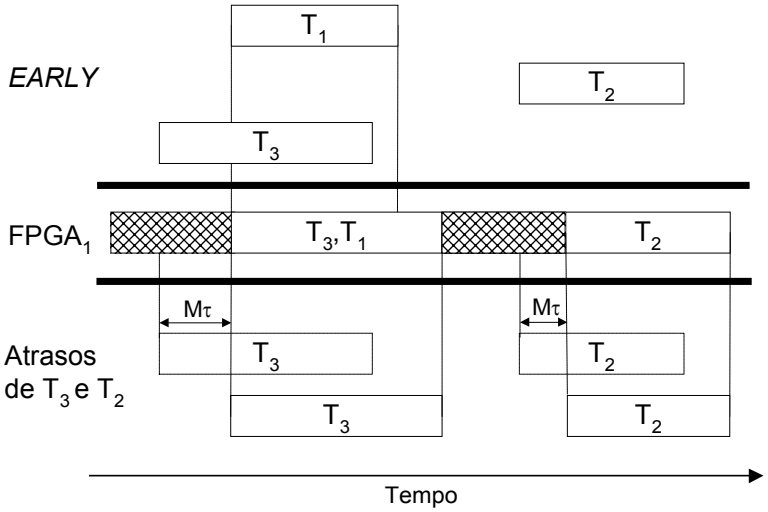
A tarefa no topo da lista é denominada *tarefa corrente* e deve ser escalonada na seqüência. Caso necessite redundância, seu próximo PDF não-coberto é denominado *PDF corrente* e o seu escalonamento é iniciado. Caso contrário, o algoritmo passa diretamente à etapa 8, para o escalonamento da tarefa.

A ordenação da lista de tarefas prontas com base no escalonamento *EARLY* de suas componentes tem por objetivo evitar que os FPGAs tenham tarefas redundantes e reconfigurações escalonadas em tempos paralelos aos intervalos *EARLY* de tarefas do sistema ainda não escalonadas. Esse tipo de situação acaba por inviabilizar a obtenção de um escalonamento válido para o *cenário corrente* e conseqüentemente o emprego da *alocação corrente* para redundância coletiva. Embora ainda não tenham sido descritos os procedimentos de escalonamento de PDFs e tarefas, é possível ilustrar essa idéia de maneira simples.

Considere um escalonamento *EARLY* de três tarefas quaisquer (T_1, T_2, T_3), tal como mostrado na Figura 6.13a. Admita que, com o objetivo de escalonar essas tarefas num dado FPGA ($FPGA_1$), através de um procedimento baseado em lista de tarefas prontas, seja definida uma prioridade que ordena a lista da seguinte forma: $\{T_1, T_2, T_3\}$. A tarefa T_1 é escalonada após uma reconfiguração de $FPGA_1$, sem atrasos, como mostrado na Figura 6.13a. Admitindo que não há recursos suficientes no FPGA para configurar simultaneamente T_1 e T_2, essa última é escalonada após a reconfiguração do dispositivo, novamente sem sofrer atrasos (Figura 6.13a). A última tarefa tomada é T_3, que não pode ser escalonada em seu intervalo *EARLY* por este ocupar parte do tempo da primeira reconfiguração de $FPGA_1$. Admitindo que há recursos para a configuração simultânea de T_1 e T_3, utiliza-se a margem de tempo da tarefa para que sua execução seja atrasada para depois da primeira reconfiguração. Porém o problema ressurge, agora com a segunda reconfiguração de $FPGA_1$ (Figura 6.13a). Esse procedimento não conseguiria portanto



a) Ordenação $\{T_1, T_2, T_3\}$



b) Ordenação por *Est* :$\{T_3, T_1, T_2\}$

Figura 6.13 Prioridade de Ordenação: Exemplo

fornecer um escalonamento válido para as tarefas na alocação disponível, ou seja, $FPGA_1$. Uma solução para contornar isso seria: escalonar T_3 após o atraso, ampliando a primeira configuração; invalidar o escalonamento de $FPGA_1$ desse ponto em diante; tentar novamente escalonar T_2. Essa alternativa, com tarefas sendo recolocadas na lista e escalonamentos sendo invalidados,

representaria um custo computacional bastante elevado, talvez até inviável, sobretudo em sistemas com várias tarefas interdependentes.

Caso as tarefas sejam tomadas para escalonamento na ordem crescente de seus *Est*, ou seja, primeiro T_3, depois T_1 e por último T_2, é possível a obtenção do escalonamento válido apresentado na Figura 6.13b. T_3 é atrasada para iniciar após a primeira reconfiguração de $FPGA_1$. T_1 é escalonada sem atrasos, sendo configurada junto com T_3. Por não haver recursos para as três tarefas simultaneamente no dispositivo, T_2 é atrasada e escalonada após uma segunda reconfiguração de $FPGA_1$. O escalonamento da Figura 6.13b é portanto obtido, sem o re-escalonamento de tarefas ou a invalidação de escalonamentos de FPGAs.

Restrição de Uso de Margem (Etapa 6)

Quando um PDF é considerado para escalonamento, a margem da *tarefa corrente* que pode ser utilizada para deslocá-lo, visando sua cobertura, deve ser limitada por algum critério. Pelo fato do algoritmo seguir um escalonamento baseado em lista de tarefas prontas, os grafos são percorridos de cima para baixo, possibilitando que o escalonamento de PDFs pertencentes às primeiras tarefas escalonadas consuma totalmente suas margens. Deve-se evitar isso porque, embora estejam associadas individualmente as tarefas, como já dito, as margens são recursos compartilhados.

Nessa etapa é definida a parte da margem da *tarefa corrente* que pode ser empregada no escalonamento do *PDF corrente*. Denominada margem utilizável ($M\tau_{UTIL}$), ela é calculada com base nas tarefas dependentes da *tarefa corrente* que também necessitam de redundância. Seu valor é dado por:

$$M\tau_{UTIL}(T_i) = \frac{M\tau_{total}(T_i)}{m+1}, \tag{6.5}$$

onde:

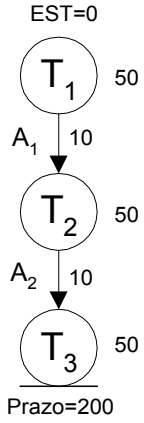
$m \rightarrow$ número de tarefas dependentes de T_i com necessidade de redundância;

$M\tau_{total}(T_i) \rightarrow$ margem total de T_i.

Isso tem por objetivo permitir que tarefas interdependentes, ocupando diferentes níveis de um mesmo grafo, possuam condições semelhantes de cobertura de PDFs. Para exemplificar essa idéia é utilizado o grafo da Figura 6.14a. Admita que todas as tarefas necessitam de redundância, o que implica no possível atraso de seus intervalos *EARLY* durante seus escalonamentos. Através dos tempos de execução e comunicação no grafo (à direita dos nós e arcos) e considerando um mapeamento de tarefas em PEs exclusivos, são obtidos os intervalos *EARLY* e *LATE* das tarefas. O primeiro é atualizado sempre que necessário durante o escalonamento, enquanto o segundo permanece constante ao longo desse procedimento. Suas margens totais iniciais e número de dependentes com necessidade de redundância também são mostrados na Figura 6.14b.

Se fosse permitida a utilização de $M\tau_{total}(T_1)$ no escalonamento de T_1 (e assim fosse feito), os intervalos *EARLY* de T_2 e T_3 deveriam ser atualizados para (90,140) e (150,200). Com esses valores, $M\tau_{total}(T_2)$ e $M\tau_{total}(T_3)$ tornariam-se zero e suas execuções não poderiam ser atrasadas durante seus escalonamentos, comprometendo diretamente a estratégia empregada pelo RECASTER na busca por uma alocação com o mínimo de FPGAs e de reconfigurações atuando como redundância coletiva do sistema, que será detalhada na descrição da próxima etapa.

Caso o uso de $M\tau_{total}(T_1)$ no escalonamento de T_1 seja limitado à $M\tau_{UTIL}(T_1) = \frac{30}{2+1}$, são asseguradas para T_2 e T_3, no mínimo, $M\tau_{total}(T_2)$=20, $M\tau_{UTIL}(T_2)$=10, $M\tau_{total}(T_3)$=10 e $M\tau_{UTIL}(T_3)$=10.



a)

Tarefa	T_1	T_2	T_3
EARLY Inicial	(0,50)	(60,110)	(120,170)
LATE	(30,80)	(90,140)	(150,200)
$M\tau_{total}(T_i)$ Inicial	30	30	30
m	2	1	0

b)

Figura 6.14 Exemplo de Restrição de Uso de Margem

```
ESCALONAMENTO_PDF (Tarefa, PDF, Alocação)
    Ordena Unidades da Alocação na Ordem Crescente dos Maiores Términos em Configurações;
    para cada Unidade da Alocação
        se (PDF Coberto)
            sai do loop;
        senão
            COBRE_PDF (Tarefa, PDF, Unidade);
```

Figura 6.15 Exemplo de Restrição de Uso de Margem

Escalonamento de PDF (Etapa 7)

Após calculada $M\tau_{UTIL}$ para o escalonamento do *PDF corrente*, é iniciada a etapa na qual é realizada (ou tentada) sua cobertura através de um FPGA da *alocação corrente*. De início, é considerada a unidade com escalonamento parcial de menor duração. Caso não seja possível a cobertura através dela, as demais são tomadas na ordem crescente dos instantes de término de seus escalonamentos parciais. Cada unidade tem uma estrutura de dados na qual são anotadas separadamente informações referentes às suas reconfigurações e configurações. Ambas têm identificadores (*Unidade.(Re)Configurações.I*), inícios (*Unidade.(Re)Configurações.I.Início*) e términos (*Unidade.(Re)Configurações.I.Término*). Nas configurações são anotadas ainda as quantidades disponíveis de recursos programáveis.

A Figura 6.15 mostra o procedimento responsável por tomar as unidades da alocação de FPGAs para a cobertura do *PDF corrente*. Caso o PDF esteja coberto, o que só ocorre após um retorno do procedimento COBRE_PDF, seu *loop* é interrompido e o algoritmo segue para a etapa seguinte. O procedimento COBRE_PDF, mostrado na Figura 6.16, é o responsável pela cobertura do *PDF corrente.* Sempre que chamado, apenas uma de suas três partes principais é executada, de acordo com a situação relativa entre o PDF e o escalonamento parcial do FPGA. São três essas situações:

1) **O PDF está contido num intervalo de reconfiguração.** Quando ocorre, havendo margem suficiente ($M\tau$), o intervalo EARLY da *tarefa corrente* deve ser atrasado até que o PDF ocorra imediatamente após o término da reconfiguração, ou seja, dentro de um intervalo de configuração. Isso porém só será realizado se outros PDFs da tarefa não forem "descobertos" pelo deslocamento. Na Figura 6.17, acima da linha cheia (parte A), é apresentado o escalona-

```
COBRE_PDF (Tarefa, PDF, Unidade)
    se (PDF⊂ Unidade.Reconfigurações.X) // 1ª Situação
        se (MτUTIL(Tarefa) ≥ Unidade.Reconfigurações.X.Término – PDF)
            Mτ = Unidade.Reconfigurações.X.Término – PDF;
            se (PDFs Anteriores da Tarefa Permanecerem Cobertos Após Atraso)
                Atrasa Escalonamento EARLY de Tarefa em Mτ;
                COBRE_PDF (Tarefa, PDF, Alocação);
                se (PDF Descoberto)
                    Restaura Escalonamento EARLY de Tarefa;
    senão
        se  (PDF ⊂ Unidade.Configurações.Y) // 2ª Situação
            se (Há Recursos em Unidade.Configurações.Y para Acrescentar Tarefa)
                se (Unidade.Configurações.Y.Término ≤ PDF+Δw) ou (Unidade.Configurações.Y.Término é
                o Maior em Unidade.Configurações)
                    Acrescenta Tarefa à Unidade.Configurações.Y; //PDF Coberto;
            se (PDF não Coberto)
                se (MτUTIL(Tarefa) ≥ Unidade.Configurações.Y.Término – PDF)
                        Mτ = Unidade.Configurações.Y.Término – PDF;
                        se (PDFs Anteriores de Tarefa Permanecem Cobertos Após Atraso)
                            Atrasa Escalonamento EARLY de Tarefa em  Mτ;
                            COBRE_PDF (Tarefa, PDF, Alocação);
                            se (PDF Descoberto)
                                Restaura Escalonamento EARLY de Tarefa;
        senão // 3ª Situação
            Unidade.Configurações.Z.Término é o Maior em Unidade.Configurações;
            se (Há Recursos em Unidade.Configurações.Z  para Acrescentar Tarefa)
                Acrescenta Tarefa à Unidade.Configurações.Z;  //PDF Coberto
            senão
                se (PDF - Unidade.Configurações.Z.Término  ≥ TREC)
                    Acrescenta à Unidade Reconfigurações.Z+1 e Configurações.Z+1 (PDF Coberto);
                senão
                    se (MτUTIL(Tarefa) ≥ (Unidade.Configurações.Z.Término - PDF + TREC))
                        Mτ = Unidade.Configurações.Z.Término – PDF+ TREC;
                        se (PDFs Anteriores da Tarefa Permanecem Cobertos Após Atraso)
                            Atrasa Escalonamento EARLY de Tarefa em  Mτ;
                            COBRE_PDF (Tarefa, PDF, Alocação);
```

Figura 6.16 Exemplo de Restrição de Uso de Margem

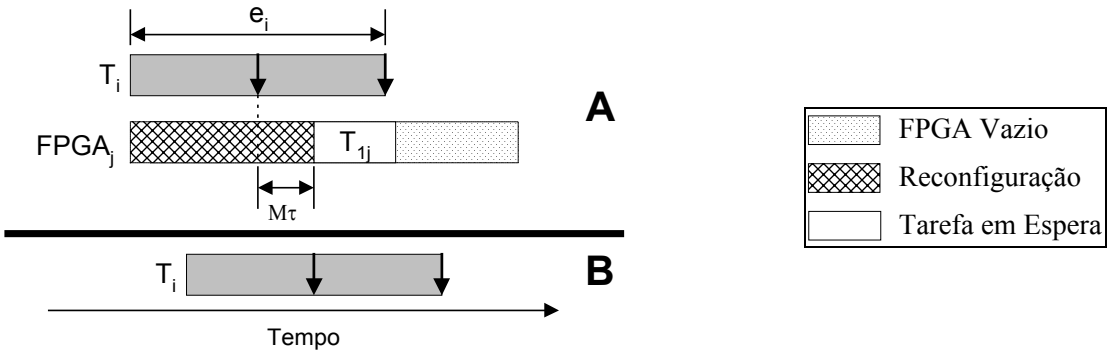


Figura 6.17 Primeira Situação Relativa PDF → FPGA

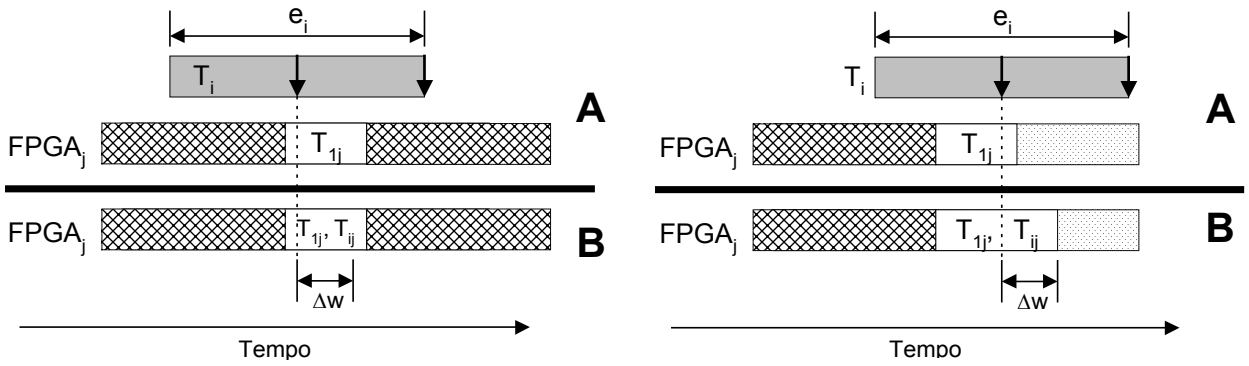
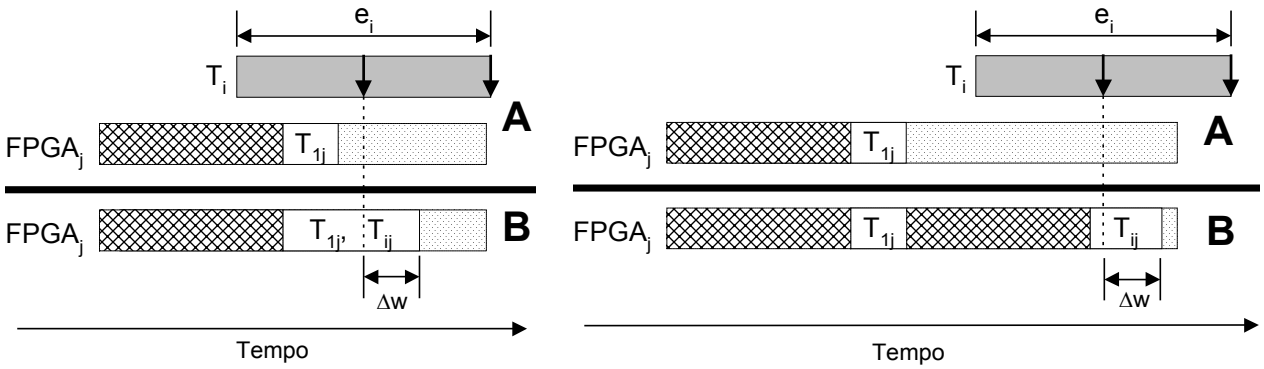


Figura 6.18 Segunda Situação Relativa

mento da uma unidade ($FPGA_j$) e o *EARLY* da tarefa corrente (T_i) antes de qualquer alteração. Abaixo dessa linha (parte B) é mostrado o escalonamento de T_i após o atraso de $M\tau \leq M\tau_{UTIL}(T_i)$. Esse atraso não implica na cobertura do PDF, a qual é tentada novamente chamando-se COBRE_PDF. Caso a execução do algoritmo não retorne desse procedimento com o PDF coberto, o escalonamento EARLY da tarefa anterior ao atraso é restaurado.

2) **O PDF está contido num intervalo de configuração.** Para que a *tarefa corrente* seja configurada nesse intervalo é importante que alguns requisitos sejam satisfeitos. O primeiro é a disponibilidade de recursos para isso. Duas ou mais tarefas podem ser configuradas simultaneamente num mesmo intervalo se: a) a soma de suas taxas de utilização for menor que *Ref %*, um valor limite para a ocupação dos seus recursos programáveis; b) a soma dos seus *# de pinos de interface* for menor que o total de pinos programáveis do dispositivo. Satisfeitas essas condições, é importante que seja verificado se o intervalo [PDF, PDF +Δw], necessário para a cobertura do *PDF corrente*, está disponível para que a *tarefa corrente* permaneça configurada nele. O PDF será portanto coberto se o instante PDF+Δw for menor que o término do intervalo de configuração (Figura 6.18a) ou se, independente disso, a configuração for a última do FPGA (Figura 6.18b). Caso não ocorra nenhuma dessas situações, tenta-se atrasar a tarefa para que o PDF ocorra imediatamente após o término da configuração, através de ações idênticas às realizadas na primeira situação relativa. Esse atraso levará o PDF a ocorrer ou num intervalo de reconfiguração ou num instante de escalonamento vazio, e o procedimento COBRE_PDF chamado após isso executará de acordo com essa nova situação relativa.



a) Caso 1 b) Caso 2

Figura 6.19 Terceira Situação Relativa

3) **O PDF ocorre num instante de escalonamento vazio.** O intervalo necessário para sua cobertura está livre no dispositivo, o que no entanto não assegura isso. Havendo recursos para configurar a tarefa na última configuração do dispositivo, a cobertura do PDF é realizada como na Figura 6.19a. Caso contrário, é verificada a possibilidade de ser escalonada uma reconfiguração antes do PDF, o que, sendo possível, é realizado como mostrado na Figura 6.19b. Não sendo possível, é verificada a possibilidade da tarefa ser atrasada para que o PDF ocorra após uma nova reconfiguração. O atraso não pode "descobrir" PDFs anteriores da tarefa.

Após o atraso, a cobertura do PDF está assegurada na próxima chamada do procedimento COBRE_PDF, daí a ausência de um procedimento de restauração do escalonamento EARLY da tarefa, em caso de retorno sem ter sido efetuada a cobertura. A Figura 6.20 mostra as duas etapas realizadas para a cobertura: atraso e escalonamento de reconfiguração-configuração.

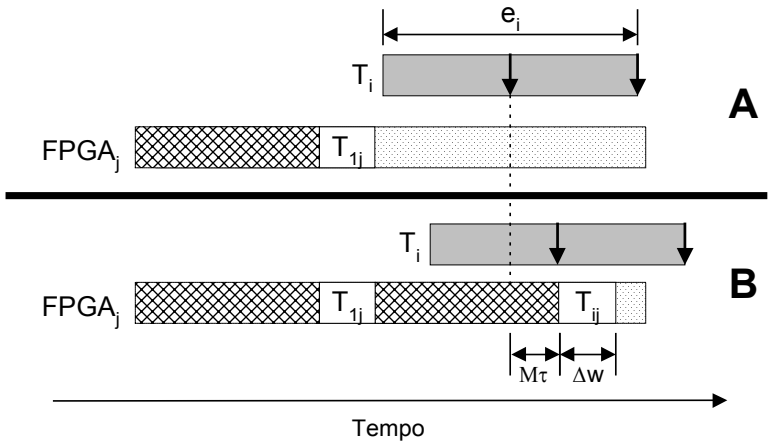


Figura 6.20 Cobertura de PDF em 2 Etapas

Escalonamento de Tarefa (Etapa 8)

Essa etapa consiste no escalonamento da *tarefa corrente* no seu intervalo *EARLY*, tendo sido atrasado ou não. Caso o *PDF corrente* tenha sido coberto, o algoritmo passa à etapa 9 do fluxograma da Figura 6.6. Caso contrário, havendo ainda alocações não-exploradas na lista, a etapa 1 é novamente executada. Se todas as alocações da lista tiverem sido exploradas, a execução do RECASTER é interrompida e o usuário informado que não foi possível a obtenção de uma solução com base na biblioteca de FPGAs fornecida.

Atualização de Margens e Lista de Tarefas (Etapa 9)

Caso a *tarefa corrente* tenha sido atrasada $M\tau$ durante o escalonamento do *PDF corrente*, todas as suas dependentes, bem como as tarefas que compartilham PEs com elas, podem ter seus intervalos de execução *EARLY* atrasados. As margens de tempo são atualizadas pelo emprego dos novos *Est* na equação 6.4.

São adicionadas à lista de tarefas prontas todas aquelas cujas dependências de comunicação/controle estejam satisfeitas. Apenas quando todos os PDFs da *tarefa corrente* são cobertos ela é retirada da lista. Isso significa que uma mesma tarefa torna-se a *tarefa corrente* do procedimento de escalonamento um número de vezes igual ao total de PDFs que possui.

Se, após atualizada, a lista de tarefas tornar-se vazia, o escalonamento do *cenário corrente* terá sido concluído com sucesso. Havendo outros cenários para serem escalonados, a etapa 2 é novamente executada. Caso contrário, todos os cenários terão sido escalonados e a execução do algoritmo passa a fase 3. Uma lista não-vazia implica na existência de PDFs ou mesmo tarefas não escalonadas. Caso isso seja constatado, a execução da fase 2 é retomada a partir da etapa 5.

6.2.3 Fase 3: Apresentação dos Resultados

A última fase de execução do algoritmo é a responsável pela geração de suas saídas no formato de arquivos de texto. Seu fluxograma é apresentado na Figura 6.21.

São fornecidos ao projetista do sistema, para cada cenário de falha, um arquivo contendo o escalonamento das tarefas nos PEs e outro contendo o escalonamento dos FPGAs, constituído

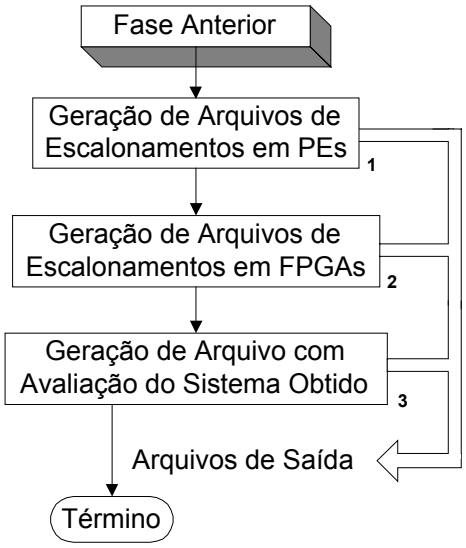


Figura 6.21 Fluxograma da Fase 3

por reconfigurações, tarefas em espera ou tarefas em execução.

Um arquivo com métricas de avaliação também é gerado. Nesse arquivo são fornecidos:

1) **a violação máxima percentual de prazo, durante o período no qual ocorre uma falha no sistema, imposta pela re-execução de uma tarefa.** A violação associada à uma tarefa é relativa ao seu prazo, caso ele esteja explícito. Do contrário, é utilizado o menor dos prazos entre suas dependentes. Os prazos empregados devem ser os associados às primeiras execuções dos grafos de especificação do sistema.

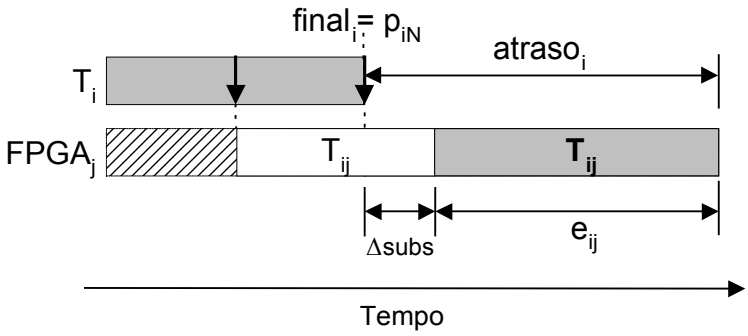
Violações são calculadas com base nos seguintes parâmetros:

a) atraso no término da execução da tarefa → diferença entre seu instante final de execução no funcionamento normal e o instante final de execução da sua tarefa redundante, considerando uma sinalização de falha em seu último PDF. Esse atraso, ilustrado na Figura 6.22 para T_i, é dado por:

$$atraso_i=(p_{iN} + \Delta_{subs} + e_{ij}) - final_i \quad (6.6)$$

b) margem residual da tarefa → a margem total da tarefa menos o valor utilizado para atrasar sua execução durante o escalonamento de seus PDFs:

$$M\tau_{res}(T_i) = M\tau_{total}(T_i) - M\tau \quad (6.7)$$



$p_{iN} \rightarrow$ último PDF de T_i;
$\Delta_{subs} \rightarrow$ tempo para detecção de falha e substituição de T_i por T_{ij};
$e_{ij} \rightarrow$ tempo de execução de T_{ij};
$final_i \rightarrow$ instante final de execução de T_i.

Figura 6.22 Atraso no Término de Tarefa Falha

Conforme explicado no capítulo anterior, durante o período no qual ocorre a falha de uma tarefa todas as suas dependentes executam imediatamente após satisfeitas suas dependências de comunicação/controle. Nessas condições, a margem residual da tarefa falha corresponde à quantidade de tempo que seu término ainda pode ser atrasado sem que nenhum prazo do sistema seja violado. Caso o atraso imposto a tarefa pela falha seja maior que sua margem residual, a falha implicará numa violação igual a diferença entre seus valores. Para uma tarefa T_i tem-se:

$$\text{Violação}_i = \text{atraso}_i - M\tau_{res}(T_i) \tag{6.8}$$

Valores menores que zero são desconsiderados e faz-se $\text{violação}_i=0$. A violação percentual é dada por:

$$\text{Violação\_Percentual}_i = \frac{\text{Violação}_i}{\text{prazo}_k^0} \text{x}100\%, \tag{6.9}$$

sendo k o índice da tarefa com menor prazo entre as dependentes de T_i. Ocorre k=i quando T_i tem prazo explícito. Na expressão 6.9 evidencia-se que o prazo da tarefa T_k empregado deve ser o correspondente à sua primeira execução. Como ficará mais claro na seção seguinte, há um valor de margem associado à cada execução de tarefa do sistema. Assim, tratando-se com sistemas multi-taxa, além do índice da tarefa, deve ser evidenciado nas expressões 6.4-6.9 seu número de cópia (Nc). Utiliza-se prazo_k^0 em 6.9 pelo fato do valor fornecido na expressão 6.8 corresponder

à violação absoluta ocorrida numa execução do grafo, que inicia-se a cada intervalo de tempo igual ao seu período.

Toda tarefa com redundância em FPGA tem sua violação percentual calculada. A violação percentual máxima fornecida no arquivo de métricas de avaliação é o maior entre esses valores.

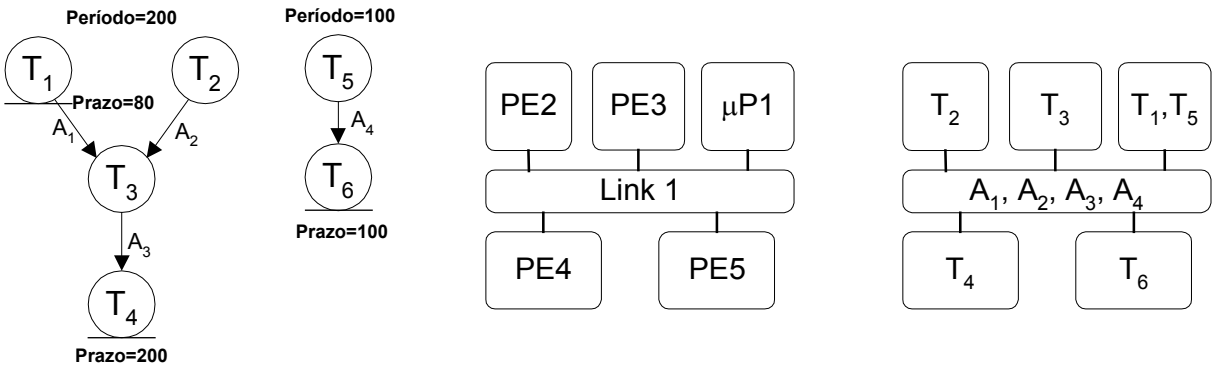
2) **a violação percentual média de prazos do sistema**. Média aritmética das violações percentuais das tarefas com redundância em FPGA;

3) **Custo total da alocação de FPGAs.** Soma dos custos unitários dos FPGAs da alocação utilizada como redundância coletiva;

4) **Área total da alocação de FPGAs.** Soma das áreas dos encapsulamentos dos FPGAs da alocação utilizada como redundância coletiva.

6.3 Exemplo

Nesta seção é apresentada, passo a passo, a aplicação do RECASTER na síntese de uma estrutura de redundância coletiva para um sistema embutido multi-taxa. O sistema é apresentado na Figura 6.23, composta por sua especificação funcional (a), sua alocação de PEs (b) e o mapeamento de suas tarefas nessa alocação (c).



a) Especificação Funcional b) Alocação c) Mapeamento

Figura 6.23 Sistema Embutido: Exemplo

Para a construção do exemplo foram consideradas as seguintes hipóteses sobre os dados de entrada do algoritmo:

1. Reconfigurações ocupam um intervalo de tempo igual ao menor tempo de execução entre as tarefas do sistema;
2. O tempo de espera (Δw) para PDFs corresponde a 1/6 do tempo de reconfiguração do FPGA empregado;
3. Os FPGAs têm recursos para, no máximo, a configuração simultânea de duas tarefas;
4. As tarefas com requisito de redundância apresentam PDFs em 50% e 100% de seus intervalos de execução.

A primeira hipótese restringe os tempos de reconfiguração dos FPGAs à ordem de grandeza dos tempos de execução das tarefas do sistema, uma das maiores restrições no emprego de redundância coletiva em um sistema embutido. A segunda hipótese relaciona o tempo para a substituição de uma tarefa falha com o tempo de configuração do FPGA responsável por sua re-execução. A relação entre esses valores foi escolhida de modo a facilitar a construção e ilustração dos exemplos desta seção, o mesmo ocorrendo com os valores das outras hipóteses.

É importante lembrar que nenhuma das hipóteses anteriores é feita sobre o problema da síntese de redundância, e sim sobre a instância empregada no exemplo ilustrativo. A seguir são descritos os procedimentos realizados em cada fase do algoritmo para esse exemplo.

6.3.1 Fase 1

- A biblioteca de FPGAs é construída com os dados apresentados nas Tabelas 6.1 e 6.2. A primeira é composta por dados relacionados exclusivamente aos dispositivos, enquanto a segunda apresenta informações sobre as implementações das tarefas do sistema nos FPGAs da biblioteca;

- As listas de tarefas do sistema são formadas com os dados apresentados na Tabela 6.3. Esses dados capturam as informações apresentadas graficamente através da Figura 6.23, além de outras como a necessidade de redundância das tarefas. As listas de PDFs não são mostradas na tabela por terem sido definidas previamente, através da quarta hipótese sobre o exemplo;

- Com os períodos dos grafos calcula-se o hiperperíodo do sistema, igual a 200 UT;

Durante o hiperperíodo, ocorre uma execução do grafo 1 e duas do grafo 2. Esse último deve ter suas tarefas duplicadas, de modo que sejam separadamente escalonadas pelo algoritmo. A segunda execução de T_5, T_5^1, tem EST igual a $EST_5^1 = EST_5^0 + 1 \cdot 100 = 100$ UT, já que $EST_5^0 = 0$ UT. O prazo de T_6^1 é $Prazo_6^1 = Prazo_6^0 + 1 \cdot 100 = 200$ UT. Após as duplicações passam a existir 8 execuções de tarefas para serem escalonadas ao longo do hiperperíodo do sistema;

- A lista de alocações de FPGAs é composta por 4 alocações homogêneas, {1 $FPGA_1$; 2 $FPGA_1$; 1 $FPGA_2$; 2 $FPGA_2$}. Ordenando-a por custo obtêm-se {1 $FPGA_1$; 1 $FPGA_2$; 2 $FPGA_1$; 2 $FPGA_2$}. Uma ordenação por área produz {1 $FPGA_2$; 2 $FPGA_2$; 1 $FPGA_1$; 2 $FPGA_1$}. Pelo fato do critério de otimização escolhido para o exemplo ser o custo da alocação de FPGAs, o algoritmo trabalha com a primeira ordenação;

- A lista de cenários de falhas gerada é {∅, T_2, T_3, T_4, T_6}. Isso significa que devem ser gerados um escalonamento sem execução de tarefas em FPGAs e quatro nos quais apenas uma tarefa (T_2, T_3, T_4 ou T_6) executa em FPGA.

	$FPGA_1$	$FPGA_2$
Área (cm^2)	9	4
Custo (US$)	70	100
Capacidade (UL)	1000	800
# de Bits de Configuração (Kbits)	732	732
# de Pinos Programáveis (pinos)	50	45
Freqüência Max. de Configuração (MHz)	25	20
Δw - Tempo de Espera (UT)	5	5
Quantidade Disponível (unid.)	2	2

Tabela 6.1 Dados dos FPGAs

		Tarefas					
		T1	T2	T3	T4	T5	T6
$FPGA_1$	Taxa de Ocupação (%)	sw	40	55	40	sw	55
	Tempo de Execução (UT)	sw	60	60	60	sw	40
$FPGA_2$	Taxa de Ocupação (%)	sw	40	30	40	sw	50
	Tempo de Execução (UT)	sw	70	60	70	sw	45

Tabela 6.2 Dados das Tarefas nos FPGAs

Grafo	1					2	
Período	200					100	
Tarefas	T_1	T_2	T_3		T_4	T_5	T_6
Prazo (UT)	80	-	-		200	-	100
EST (UT)	0	0	-		-	0	-
Necessidade de Redundância (Sim/Não)	Não	Sim	Sim		Sim	Não	Sim
PE	μP1	PE2	PE3		PE4	μP1	PE5
Tempo de Execução (UT)	50	50	50		50	30	30
Área (cm^2)	4	4	4		4	4	4
Número de Pinos de Interface	-	16	32		16	-	16
Arcos de Entrada	-	-	A_1	A_2	A_3	-	A_4
Link	-	-	Link1	Link1	Link1	-	Link1
Tempo de Comunicação (UT)	-	-	5	5	5	-	5
Arcos de Saída	A_1	A_2	A_3		-	A_4	-
Link	Link1	Link1	Link1		-	Link1	-
Tempo de Comunicação (UT)	5	5	5		-	5	-

Tabela 6.3 Listas de Tarefas: Dados de Entrada

6.3.2 Fase 2

Com as informações mostradas na seção anterior é possível estimar-se o número máximo de iterações que podem ou serão realizadas em cada um dos três *loops* da fase 2 do RECASTER. O *loop* de alocação pode repetir-se no máximo 4 vezes, pelo fato de existirem apenas 4 alocações para serem exploradas. Numa dessas repetições, para que possam ser obtidos escalonamentos para todos os cenários da lista, o *loop* de cenários de falhas deve ser repetido 5 vezes. Na sua primeira execução, correspondente ao cenário ∅, o *loop* de escalonamento é repetido 12 vezes, uma vez que existem 8 execuções de tarefas para serem escalonadas, das quais 5 apresentam dois PDFs. Nas demais repetições do *loop* de cenários, o *loop* de escalonamento é repetido apenas para escalonar as 8 execuções de tarefas do sistema.

Para não tornar a descrição do exemplo cansativa, apresentando várias repetições de procedimentos bastante semelhantes, optou-se por descrever a execução do algoritmo apenas para os dois primeiros cenários da lista. Na seqüência são comentadas as ações realizadas nas etapas da fase 2.

- A primeira alocação da lista, contendo apenas 1 dispositivo do tipo $FPGA_1$, é escolhida para ser explorada como redundância coletiva e definida como *alocação corrente*;

- O primeiro cenário da lista, ∅, é definido como o *cenário corrente*;

- Como no *cenário corrente* nenhuma das tarefas executa em FPGAs, todos os tempos considerados para suas execuções encontram-se na Tabela 6.3. O escalonamento ALAP produz os seguintes intervalos de execução para as tarefas do sistema:

Tarefas	T_1^0	T_2^0	T_3^0	T_4^0	T_5^0	T_6^0	T_5^1	T_6^1
Intervalo de Execução	(30,80)	(40,90)	(95,145)	(150,200)	(35,65)	(70,100)	(135,165)	(170,200)

Tabela 6.4 ALAP

Com base nos dados da Tabela 6.4 é feita a ordenação das tarefas nos PEs do sistema. Em μP1 a tarefa de maior prioridade é T_5 (primeira execução), seguida por T_1 (única execução). A ordenação em todos os PEs é apresentada na Tabela 6.5;

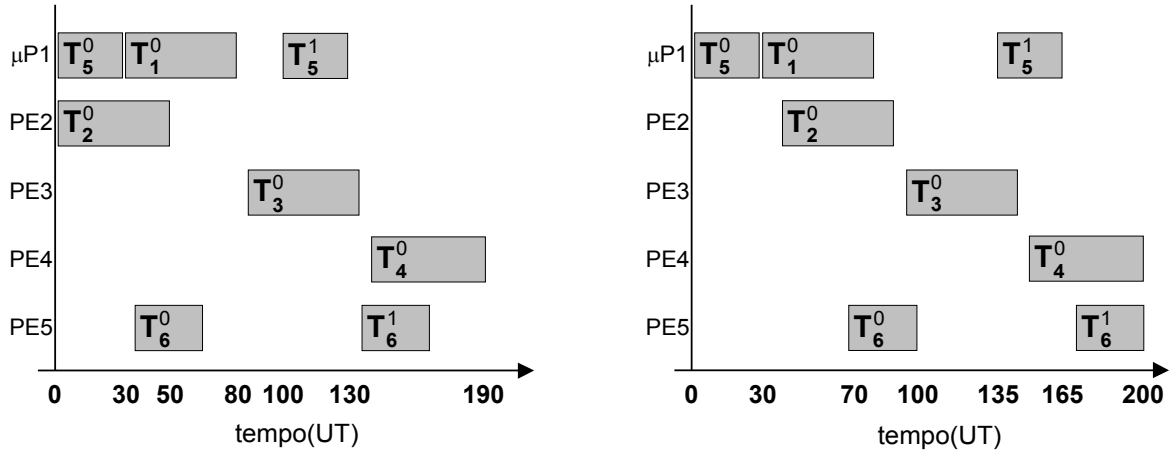
PE	μP1	PE2	PE3	PE4	PE5
Tarefas (Ordenação)	$1º \rightarrow T_5^0$; $2º \rightarrow T_1^0$; $3º \rightarrow T_5^1$	T_2^0	T_3^0	T_4^0	$1º \rightarrow T_6^0$; $2º \rightarrow T_6^1$

Tabela 6.5 Ordenação de Tarefas em PEs

- Os escalonamentos *EARLY* e *LATE*, bem como as margens iniciais das tarefas, são mostradas na Tabela 6.6. As Figuras 6.24a e 6.24b apresentam esses escalonamentos através de diagramas de tempo;

- A lista de tarefas prontas para escalonamento é iniciada com T_1^0, T_2^0, T_5^0 e T_5^1, ou seja, todas aquelas sem arcos de entrada nos grafos de especificação do sistema;

- Entre as tarefas da primeira lista apenas T_2 requer redundância, logo é a única a ter como métrica de prioridade para escalonamento o instante de ocorrência do seu menor PDF não-coberto. Todas as demais empregam o instante inicial de execução. O primeiro PDF de T_2^0 (p_{21}^0)



a) Diagrama de Tempos: *EARLY* b) Diagrama de Tempos: *LATE*

Figura 6.24 Escalonamentos *EARLY* e *LATE* : Exemplo

Tarefas	T_1^0	T_2^0	T_3^0	T_4^0	T_5^0	T_6^0	T_5^1	T_6^1
EARLY	(30,80)	(0,50)	(85,135)	(140,190)	(0,30)	(35,65)	(100,130)	(135,165)
LATE	(30,80)	(40,90)	(95,145)	(150,200)	(0,30)	(70,100)	(135,165)	(170,200)
Margens Iniciais	0	40	10	10	0	35	35	35

Tabela 6.6 *EARLY*, *LATE* e Margens das Tarefas

ocorre em 50% de seu intervalo de execução *EARLY*, ou seja, no instante 25 do hiperperíodo. A primeira lista ordenada é portanto $\{T_5^0 \rightarrow 0;\ T_2^0 \rightarrow 25;\ T_1^0 \rightarrow 30;\ T_5^1 \rightarrow 100\}$ e T_5^0 a primeira tarefa (execução de tarefa) definida como corrente;

- Por T_5 não necessitar redundância em FPGA, sua primeira execução é escalonada diretamente no intervalo (0,30) de μP1;

- Uma vez que T_5^0 não foi atrasada, não é necessária a atualização das margens do sistema. Na atualização da lista, ela é retirada e sua sucessora T_6^0 acrescentada. A nova lista é $\{T_2^0 \rightarrow 25;\ T_1^0 \rightarrow 30;\ T_5^1 \rightarrow 100;\ T_6^0 \rightarrow 50\}$ e, pelo fato de não estar vazia, o *loop* de escalonamento é repetido;

- Após ordenada, a lista torna-se $\{T_2^0 \rightarrow 25;\ T_1^0 \rightarrow 30;\ T_6^0 \rightarrow 50; T_5^1 \rightarrow 100\}$ e T_2^0 escolhida para escalonamento;

- Por T_2 necessitar de redundância em FPGA, seu PDF p_{21}^0 é definido como corrente para escalonamento. Pode-se utilizar $M\tau_{UTIL}(T_2^0) = \frac{M\tau_{total}(T_2^0)}{m+1} = \frac{40}{2+1} = 13{,}3\ UT$ nesse procedimento, já que todas as dependentes de T_2^0 têm necessidade de redundância;

O único FPGA da *alocação corrente*, $FPGA_1$, é considerado para cobrir p_{21}^0. O escalonamento parcial desse dispositivo é nulo, ou seja, está vazio durante todo o hiperperíodo do sistema. A situação relativa entre ele e p_{21}^0 é portanto a terceira (Figura 6.25a), o que implica na necessidade de se escalonar uma reconfiguração em $FPGA_1$ antes de sua ocorrência. Uma reconfiguração de $FPGA_1$ é realizada em 30 UT. Mesmo iniciando em 0 UT, terminaria depois de p_{21}^0, como mostra a Figura 6.25b. É necessário portanto que T_2^0 seja atrasada de 5 UT, o que é possível por $M\tau_{UTIL}(T_2^0)$ > 5 UT. O escalonamento parcial de $FPGA_1$ após a cobertura de p_{21}^0 é mostrado na Figura 6.21c. Caso não houvesse margem suficiente para tal procedimento, e pelo fato do escalonamento de $FPGA_1$ estar vazio, seria possível que uma reconfiguração fosse escalonada entre duas execuções consecutivas do sistema, ou seja, entre dois intervalos de hiperperíodo. Isso evitaria que T_2^0 fosse atrasada, e o escalonamento parcial de $FPGA_1$ seria o mostrado na Figura 6.25d;

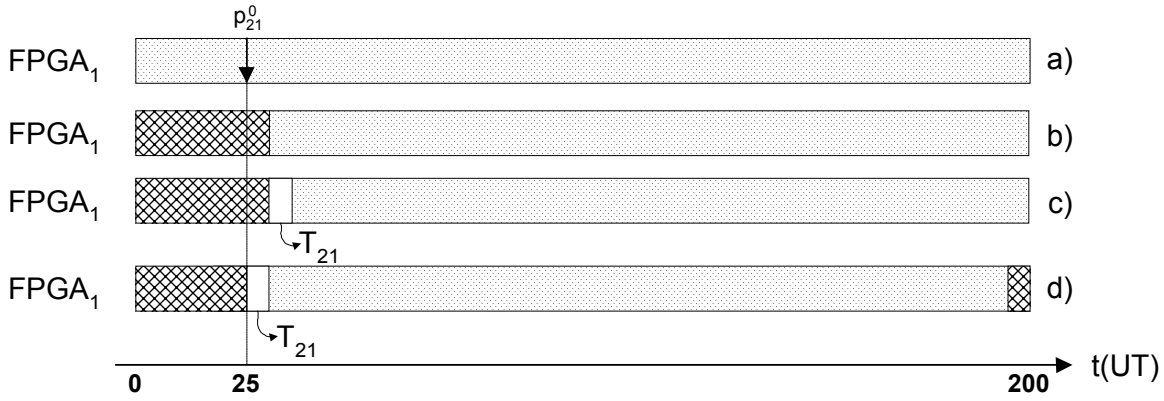


Figura 6.25 Escalonamentos Parciais de $FPGA_1$: Cobertura de p_{21}^0

- Após o atraso, T_2^0 é escalonada no intervalo (5,55), seu intervalo de execução atrasado por 5 UT. Pelo fato do *PDF corrente* ter sido coberto, o escalonamento parcial do sistema é válido;

- A atualização dos intervalos de execução *EARLY* de suas dependentes não produz alterações. Sua única dependente direta, T_3^0, inicia num instante bem posterior (85 UT) a soma do término de T_2^0 (55 UT) com o tempo de comunicação (5 UT) entre elas. Por ainda haver um PDF de T_2^0 para ser coberto (p_{22}^0), ela é mantida na lista, impedindo a inserção de T_3^0. Uma nova iteração do *loop* de escalonamento é iniciada;

- Na lista, T_2^0 é agora associada ao instante de ocorrência de p_{22}^0, que corresponde ao término de sua execução. Sua reordenação produz $\{T_1^0 \rightarrow 30;\ T_6^0 \rightarrow 50;\ T_2^0 \rightarrow 55;\ T_5^1 \rightarrow 100\}$ e T_1^0 torna-se a *tarefa corrente*;

- Por não necessitar redundância em FPGA, T_1^0 é escalonada diretamente em (30,80) e retirada da lista de tarefas durante sua atualização;

- Uma nova ordenação da lista produz $\{T_6^0 \rightarrow 50;\ T_2^0 \rightarrow 55;\ T_5^1 \rightarrow 100\}$. T_6^0 é selecionada para escalonamento;

- Toda a margem de T_6^0 poderia ser utilizada no escalonamento do seu primeiro PDF, p_{61}^0, uma vez que ela não possui dependentes. No entanto, pelo fato de existirem recursos no dispositivo para que T_{61} seja configurada junto com T_{21}, sua última configuração é estendida para cobrir p_{61}^0, ou seja, até $p_{61}^0 + \Delta w = 50 + 5 = 55$ UT. Ocorre portanto a situação mostrada na Figura 6.19[a]. A Figura 6.26b apresenta o escalonamento parcial de $FPGA_1$ após a cobertura de p_{61}^0;

- T_6^0 é escalonada no seu intervalo *EARLY*, (35,65), e mantida na lista, agora com parâmetro de ordenação equivalente ao instante de ocorrência de seu segundo PDF;

- A lista após reordenação torna-se $\{T_2^0 \rightarrow 55;\ T_6^0 \rightarrow 65; T_5^1 \rightarrow 100\}$, sendo T_2^0 novamente definida como *tarefa corrente*, agora para o escalonamento de p_{22}^0. Para esse novo escalonamen-

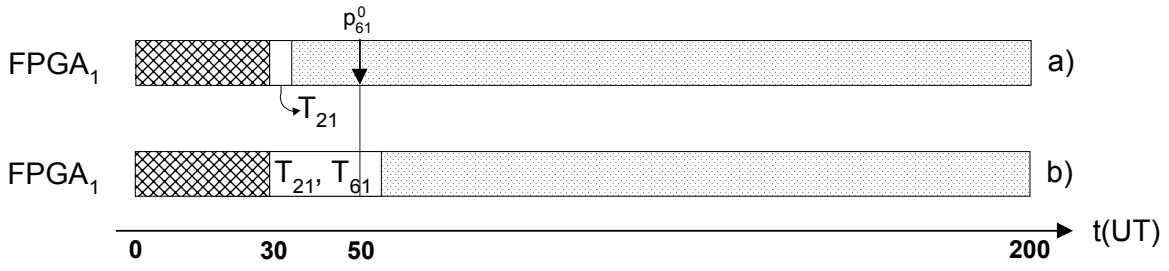


Figura 6.26 Escalonamentos Parciais de $FPGA_1$: Cobertura de p_{61}^0

to têm-se $M\tau_{UTIL}(T_2^0) = \frac{M\tau_{total}(T_2^0)}{m+1} = \frac{35}{2+1} = 11{,}6\,UT$, pelo fato de 5 UT já terem sido utilizados da margem inicial de T_2^0;

- p_{22}^0 é coberto sem a necessidade de atrasos. A última configuração de $FPGA_1$, que já contém T_{21} (Figura 6.27a), é estendida até $p_{22}^0 + \Delta w$ = 60 UT, como mostra a Figura 6.27b. Por não ter PDFs ainda descobertos, T_2^0 é retirada da lista e T_3^0 inserida;

- A lista, agora com T_3^0 ordenada pelo instante do seu menor PDF, é $\{T_6^0 \rightarrow 65;\ T_5^1 \rightarrow 100;\ T_3^0 \rightarrow 110\}$. O *PDF corrente* é p_{62}^0, coberto tal como p_{22}^0, pela extensão da última configuração de $FPGA_1$, que já contém T_{61}. A Figura 6.28 ilustra esse procedimento;

- T_6^0 é retirada da lista e T_5^1 torna-se a *tarefa corrente* da iteração seguinte do *loop* de escalonamento. Por não requisitar redundância, ela é escalonada diretamente em (100,130) e retirada da lista. Sua sucessora T_6^1 é inserida;

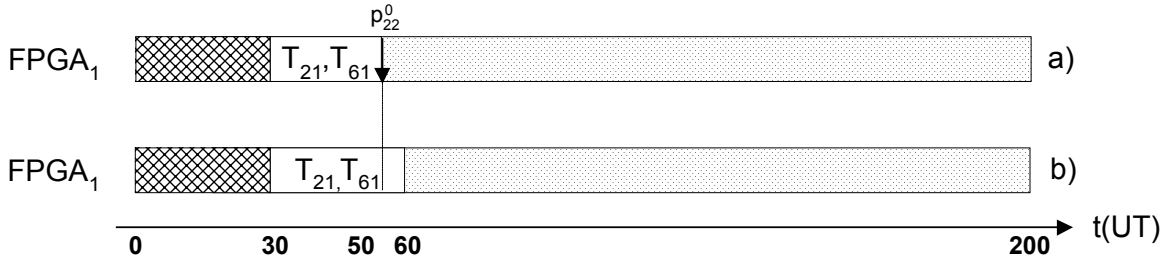


Figura 6.27 Escalonamentos Parciais de $FPGA_1$: Cobertura de p_{22}^0

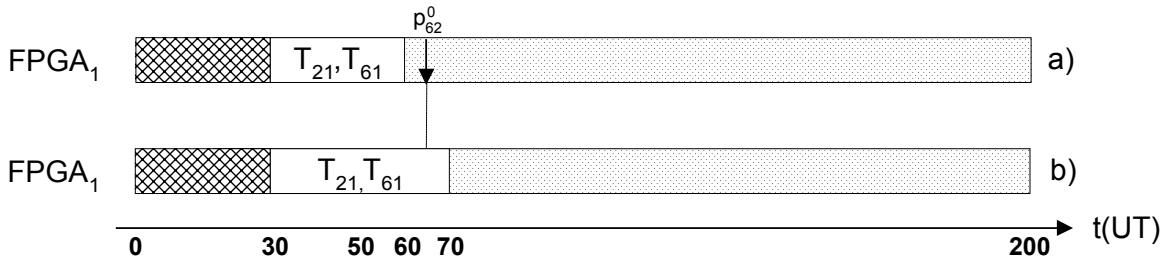


Figura 6.28 Escalonamentos Parciais de FPGA$_1$: Cobertura de p_{62}^0

- A lista ordenada torna-se $\{T_3^0 \rightarrow 110;\ T_6^1 \rightarrow 150\}$, sendo p_{31}^0 o *PDF corrente*. Cobri-lo pela extensão da última configuração de FPGA$_1$ não é possível, pois as taxas de utilização de T_{31}, T_{21} e T_{61} somam mais que 100%. Ocorre agora o caso 2 da terceira situação relativa entre PDFs e escalonamentos parciais de FPGAs, no qual é possível configurar-se o dispositivo antes do instante de ocorrência do PDF;

- Uma reconfiguração de FPGA$_1$ é escalonada em (70,100) e T_{31} mantida configurada no intervalo (100, 115), para cobrir p_{31}^0. A Figura 6.29 mostra os escalonamentos parciais de FPGA$_1$, antes e depois disso;

- T_3^0 é mantida na lista e torna-se novamente a *tarefa corrente*, para o escalonamento de seu segundo PDF. A cobertura de p_{32}^0 é realizada pela extensão da última configuração de FPGA$_1$ até 135+Δw = 140 UT, como mostrado na Figura 6.30. Após isso, T_3^0 é retirada da lista e T_4^0 inserida;

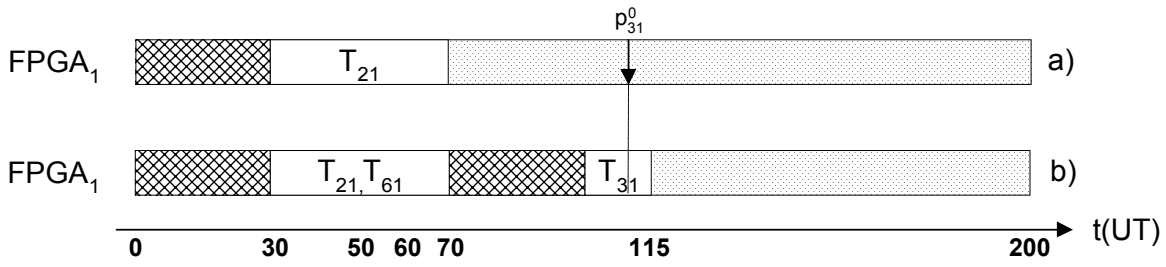


Figura 6.29 Escalonamentos Parciais de FPGA$_1$: Cobertura de p_{31}^0

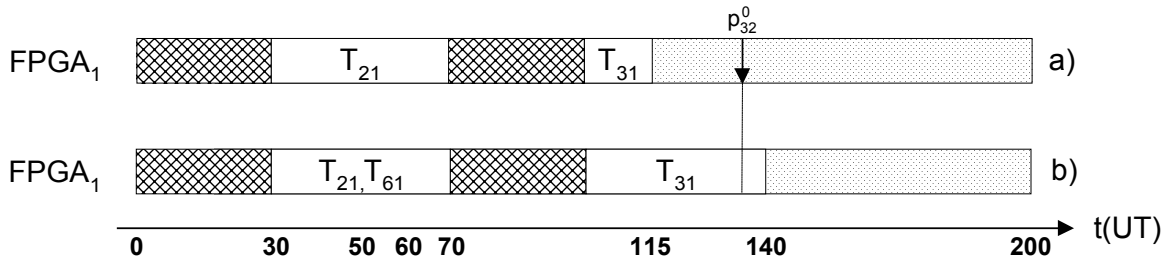


Figura 6.30 Escalonamentos Parciais de $FPGA_1$: Cobertura de p_{32}^0

- A lista torna-se $\{T_6^1 \rightarrow 150;\ T_4^0 \rightarrow 165\}$ e p_{61}^1 considerado para escalonamento. Pelo fato de não existir recursos para a configuração simultânea de T_{31} e T_{61} em $FPGA_1$, a simples extensão da sua última configuração é descartada. O intervalo de tempo entre seu término e p_{61}^1 não é suficiente para uma reconfiguração de $FPGA_1$. Entre p_{61}^1 e o escalonamento parcial de $FPGA_1$ ocorre a situação mostrada na Figura 6.20. Nesse caso, há margem suficiente para que o *PDF corrente* (na verdade a *tarefa corrente*) seja atrasado até depois de uma nova reconfiguração, escalonada para cobri-lo. p_{61}^1 é atrasado 20 UT, passando a ocorrer em 170. O valor de $M\tau_{total}(T_6^1)$ é reduzindo em 20 UT, tornando-se $M\tau_{total}(T_6^1) = 35 - 20 = 15\,UT$. As Figuras 6.31a e 6.31b mostram os escalonamentos de $FPGA_1$ antes e depois da cobertura de p_{61}^1;

- T_6^1 é escalonada em (155,185) e a lista ordenada, produzindo $\{T_4^0 \rightarrow 165;\ T_6^1 \rightarrow 185\}$. Sendo T_4^0 a *tarefa corrente* e pelo fato de necessitar de redundância, p_{41}^0 é considerado para escalonamento, podendo para isso utilizar toda a margem da tarefa, igual a 10 UT, pelo fato dela não possuir dependentes;

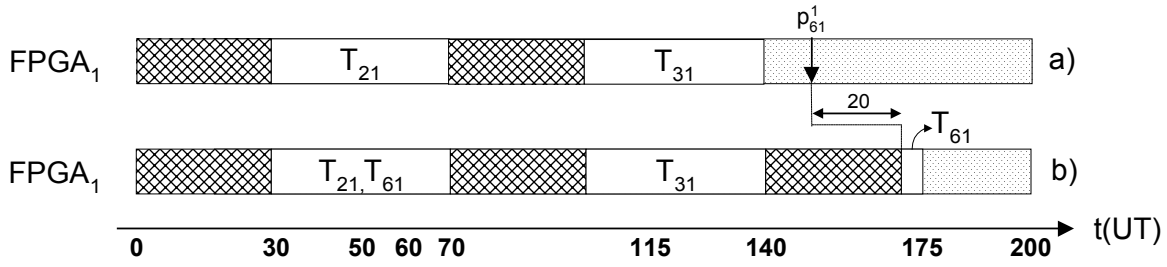


Figura 6.31 Escalonamentos Parciais de $FPGA_1$: Cobertura de p_{61}^1

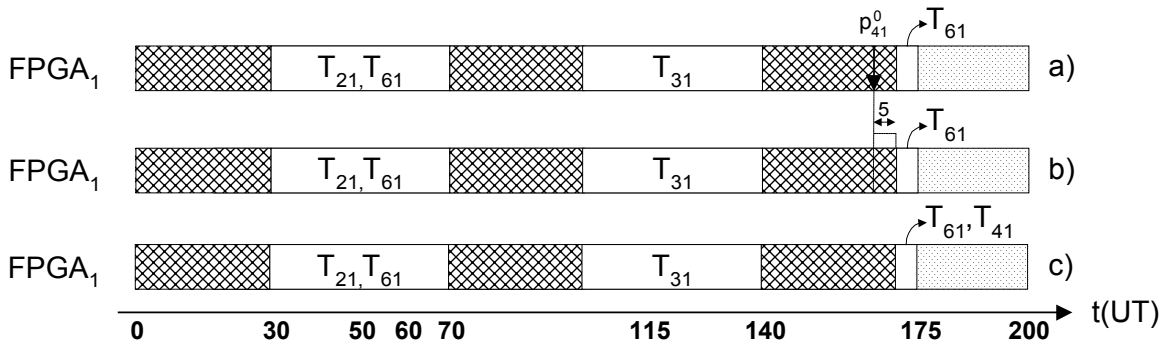


Figura 6.32 Escalonamentos Parciais de FPGA$_1$: Cobertura de p_{41}^0

- Percebe-se pela Figura 6.32a que entre o escalonamento parcial de FPGA$_1$ e p_{41}^0 ocorre a primeira situação mostrada na Figura 6.17. Por haver margem suficiente, p_{41}^0 é atrasada até o término da terceira reconfiguração de FPGA$_1$, como mostrado na Figura 6.32b. A nova situação é a da Figura 6.18, pelo fato de existir recursos em FPGA$_1$ suficientes para a configuração simultânea de T_{61} e T_{41} e o intervalo $[p_{41}^0, p_{41}^0+\Delta w]$ encontrar-se dentro do intervalo das tarefas em espera (Figura 6.32c);

- T_4^0 é escalonada em (145, 195). A lista, atualizada e reordenada, torna-se $\{T_6^1 \rightarrow 185;\ T_4^0 \rightarrow 195\}$. Na seqüência, p_{62}^1 e p_{42}^0 são cobertos, um por iteração do *loop* de escalonamento, através da extensão da última configuração de FPGA$_1$ até o término do hiperperíodo ($p_{42}^0 + \Delta w$=200 UT). Caso a última configuração não atingisse esse instante após todas as tarefas escalonadas, seu intervalo seria estendido para isso. A Figura 6.33 mostra o escalonamento final para o cenário de falhas ∅, que inclui o escalonamento das tarefas do sistema e de FPGA$_1$;

- Após o escalonamento de p_{42}^0, a lista de tarefas prontas para escalonamento torna-se vazia e o loop de cenários de falhas é repetido com a falha de T_2 como *cenário corrente*;

Uma vez que T_2 deve executar em FPGA$_1$ no *cenário corrente*, durante a determinação das margens de tempo iniciais do sistema emprega-se o tempo de execução de T_{21}, mostrado na Tabela 6.2. Após realizado o escalonamento ALAP, a ordenação de tarefas nos PEs produz resultado idêntico ao mostrado anteriormente na Tabela 6.5;

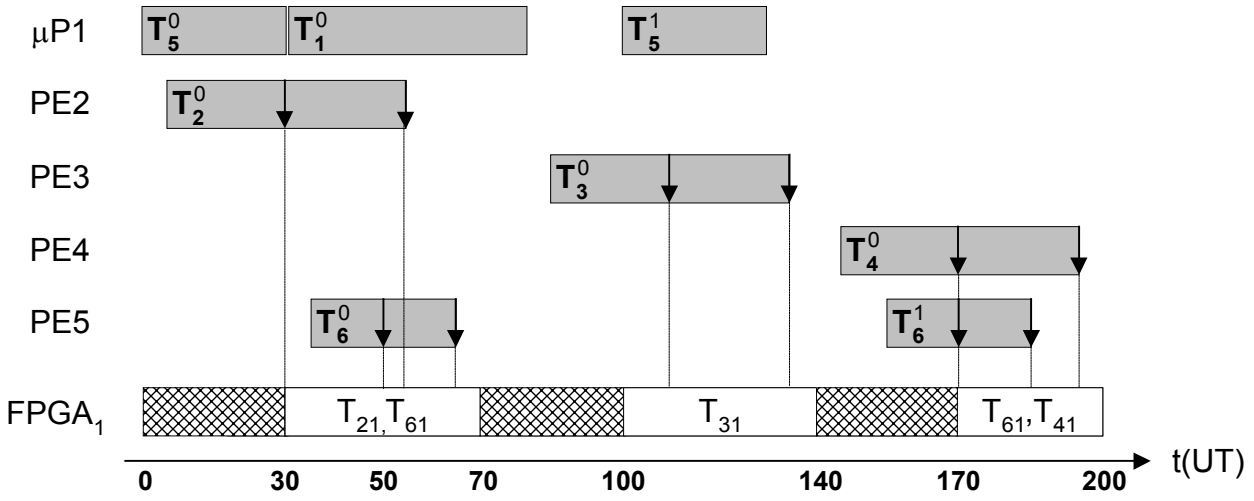


Figura 6.33 Escalonamento para o Cenário ∅

- Os escalonamentos *EARLY* e *LATE*, bem como as margens iniciais das tarefas, são mostradas na Tabela 6.7, que difere da 6.6 apenas nos dados relativos à T_2;

Tarefas	T_1^0	T_{21}	T_3^0	T_4^0	T_5^0	T_6^0	T_5^1	T_6^1
EARLY	(30,80)	(0,60)	(85,135)	(140,190)	(0,30)	(35,65)	(100,130)	(135,165)
LATE	(30,80)	(30,90)	(95,145)	(150,200)	(0,30)	(70,100)	(135,165)	(170,200)
Margens Iniciais	0	30	10	10	0	35	35	35

Tabela 6.7 *EARLY*, *LATE* e Margens das Tarefas: Cenário T_2

- A primeira lista de tarefas prontas é $\{T_1^0, T_{21}, T_5^0$ e $T_5^1\}$, tal como no cenário anterior;

- Pelo fato de nenhuma tarefa ter necessidade de redundância nesse cenário, todas têm como métrica de prioridade para escalonamento seus *Est*'s. Isso implica na lista ordenada $\{T_{21}\rightarrow 0$, $T_5^0 \rightarrow 0$ $T_1^0 \rightarrow 30$; $T_5^1 \rightarrow 100\}$, da qual T_{21} é definida como *tarefa corrente*. O empate entre T_{21} e T_5^0, caso fossem cópias com Nc distintos, seria quebrado tomando-se a com menor Nc. Quando ambas têm Nc's iguais, a tarefa escolhida é a que se encontra na lista há mais tempo;

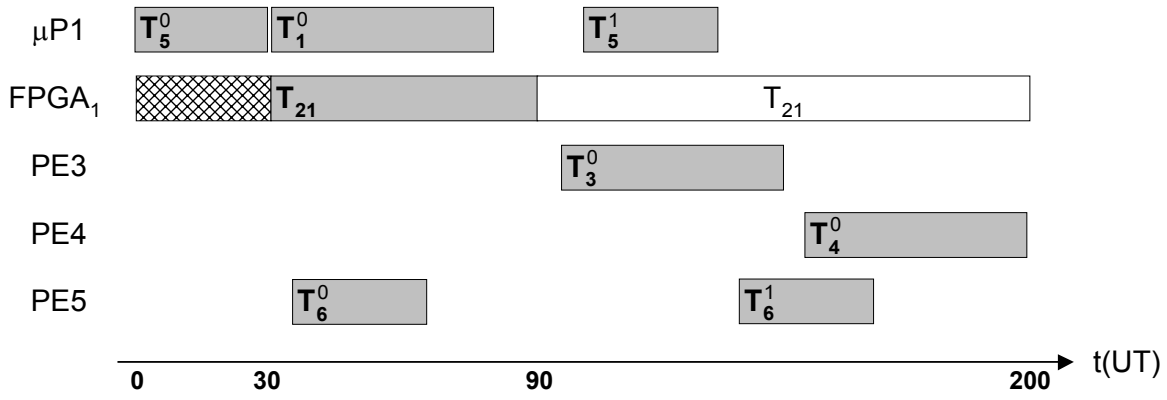


Figura 6.34 Escalonamento para o Cenário T_2

- O escalonamento de T_{21} em $FPGA_1$ deve ser realizado posteriormente a uma reconfiguração do dispositivo. Pelo fato de não haver tempo para que isso seja realizado (seu instante inicial é zero), verifica-se a possibilidade da tarefa ser atrasada. A margem total da tarefa é 30 UT e pode ser completamente utilizada nesse procedimento pelo fato de não existirem outras tarefas do sistema para serem escalonadas em FPGA nesse cenário. Assim, T_{21} é atrasada para que inicie em 30 UT;

- A atualização de margens altera o intervalo *EARLY* de T_3^0, que torna-se igual ao seu *LATE*. A margem de T_{21} passa a ser zero e ela é retirada da lista;

- As tarefas T_3^0 e T_4^0 são escalonadas nos seus intervalos *LATE* devido ao atraso imposto à T_{21}. As demais tarefas são escalonadas nos seus intervalos *EARLY* iniciais (Tabela 6.7). A Figura 6.34 mostra o escalonamento final para o cenário de falhas T_2, no qual T_{21} executa em $FPGA_1$ durante o intervalo (30,90) e permanece apenas configurada em (90,200);

- Após percorrida toda a lista de cenários de falhas, a execução do algoritmo passa à fase 3.

6.3.2 Fase 3

Nessa fase são gerados arquivos de texto contendo as informações mostradas graficamente através das Figuras 6.33 e 6.34, relativas aos escalonamentos associados aos cenários de falhas.

	p_{iN}^{Nc}	Δ_{subs}	e_{il}^{Nc}	$final_i^{Nc}$	$M\tau_{res}^{i}$	$prazo_k^0$	Violação Percentual (%)	
T_2^0	55	10	60	55	35	200	17,5	
T_3^0	135	10	60	135	10	200	30	
T_4^0	195	10	60	195	5	200	32,5	
T_6^0	65	10	40	65	35	100	15	
T_6^1	185	10	40	185	15	100	65	**Máxima**
							30,4	**Média**

Tabela 6.8 Violações de Prazo

Os valores máximo e médio das violações percentuais de prazos provocadas por falhas e re-execuções de tarefas em $FPGA_1$ (Tabela 6.8), são gravados no arquivo de avaliação.

O custo e a área total da alocação de FPGAs, também gravados no arquivo de avaliação, correspondem aos valores unitários de $FPGA_1$, sendo respectivamente 9 cm^2 e US$ 70. A execução do algoritmo é encerrada após o fechamento de todos os arquivos gerados nessa fase.

A Figura 6.35 mostra a transição do sistema entre os cenários de falhas $\varnothing$ e T_2, após ser detectada uma falha em p_{21}^1. As reconfigurações de $FPGA_1$ são suspensas e as tarefas dependentes de T_2^0 executadas imediatamente após disponíveis suas entradas. Porém isso não evita a violação de 35 UT no prazo da tarefa T_4^0, correspondente à máxima violação provocada por uma falha de T_2 (Tabela 6.8). O escalonamento do cenário T_2 permanece indefinidamente.

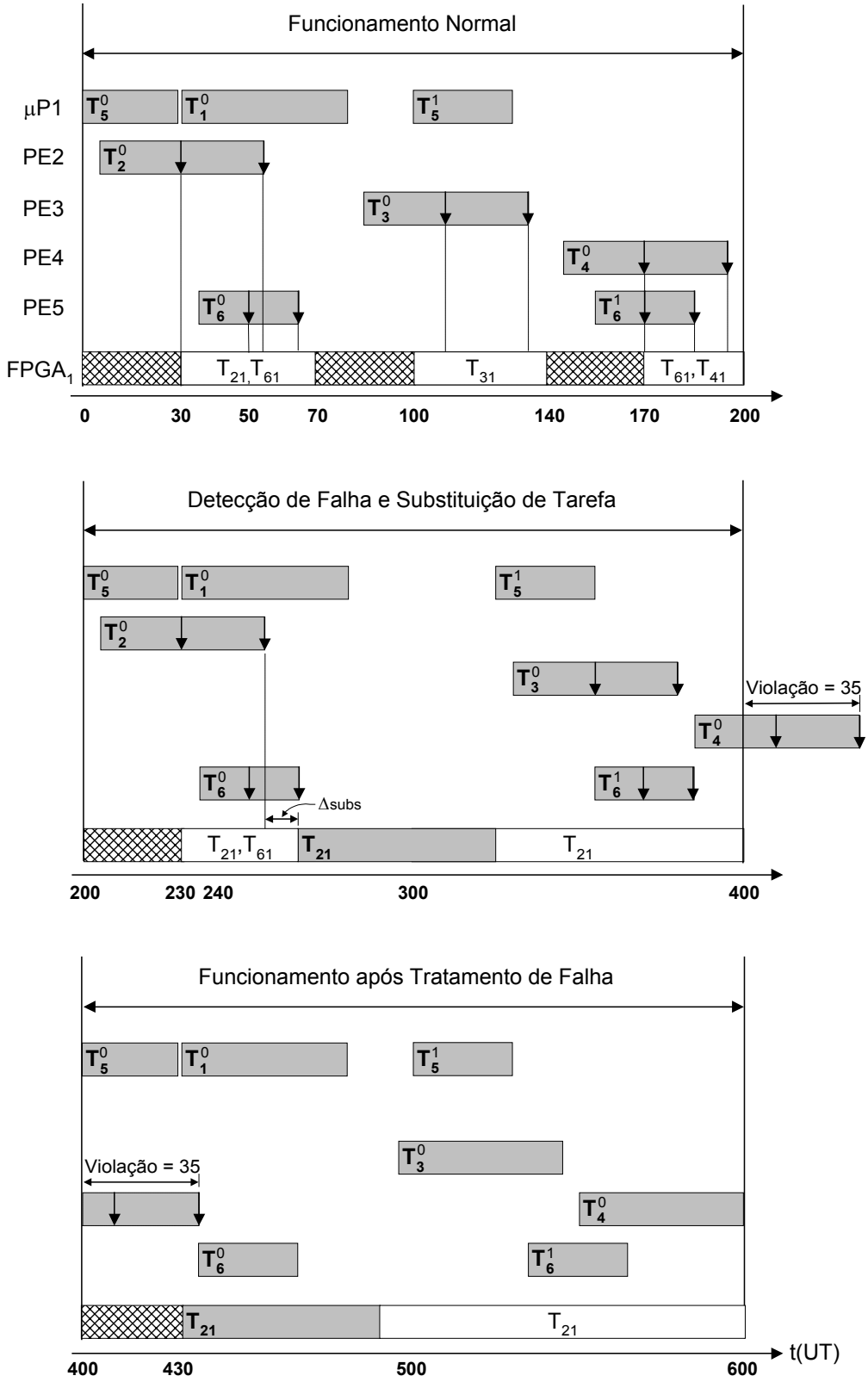


Figura 6.35 Funcionamento do Sistema

Resumo do Capítulo 6

Neste capítulo foi apresentado o algoritmo RECASTER, desenvolvido em suporte a estratégia de redundância coletiva, visando auxiliar na síntese de sistemas embutidos que a empreguem.

A apresentação do algoritmo foi realizada partindo-se de uma visão geral do seu fluxo de execução, dividido em 3 fases: Leitura de Dados e Preenchimento de Estruturas Internas; Escalonamento e Síntese de Redundância Coletiva; Apresentação dos Resultados. A descrição da fase 1 permitiu que fossem conhecidos os dados de entrada necessários para o algoritmo e como são utilizados na formação de suas principais estruturas internas, tais como grafos de tarefas, lista de alocação e de cenários de falhas.

A fase 2 é a única que pode ser executada mais de uma vez, através de repetições do *loop* de alocação. Como mostrado na sua descrição, é responsável pela busca de uma estrutura de redundância coletiva compatível com os prazos do sistema. Para cada cenário de falhas definido na fase 1, o algoritmo tenta construir um escalonamento de tarefas através de um procedimento baseado em lista de tarefas prontas. Cada tarefa (ou execução de tarefa) é tomada individualmente para ser escalonada, podendo ter seu intervalo de execução atrasado, caso necessite redundância em FPGA ou deva ser diretamente escalonada num deles. Na primeira situação, o atraso é realizado para que um de seus PDFs seja coberto, enquanto, na segunda, isso pode ocorrer para que a tarefa seja configurada no dispositivo antes de sua execução.

A terceira e última fase do algoritmo gera arquivos de texto com os escalonamentos das tarefas, redundantes ou não, em todos os cenários de falhas simples para o sistema. É gerado também um arquivo com métricas de avaliação da solução obtida, incluindo custo e área da alocação de FPGAs utilizada e violações de prazos do sistema em caso de falhas. Os dados relativos aos FPGAs estão diretamente associados aos critérios de otimização disponíveis ao projetista, enquanto as violações permitem estimar o quanto as falhas prejudicam o desempenho do sistema durante o hiperperíodo no qual ocorrem.

O capítulo é finalizado com a descrição de um exemplo ilustrativo de aplicação do RECASTER, na qual seus principais procedimentos são mostrados de maneira prática.

Capítulo 7

Avaliação do Algoritmo

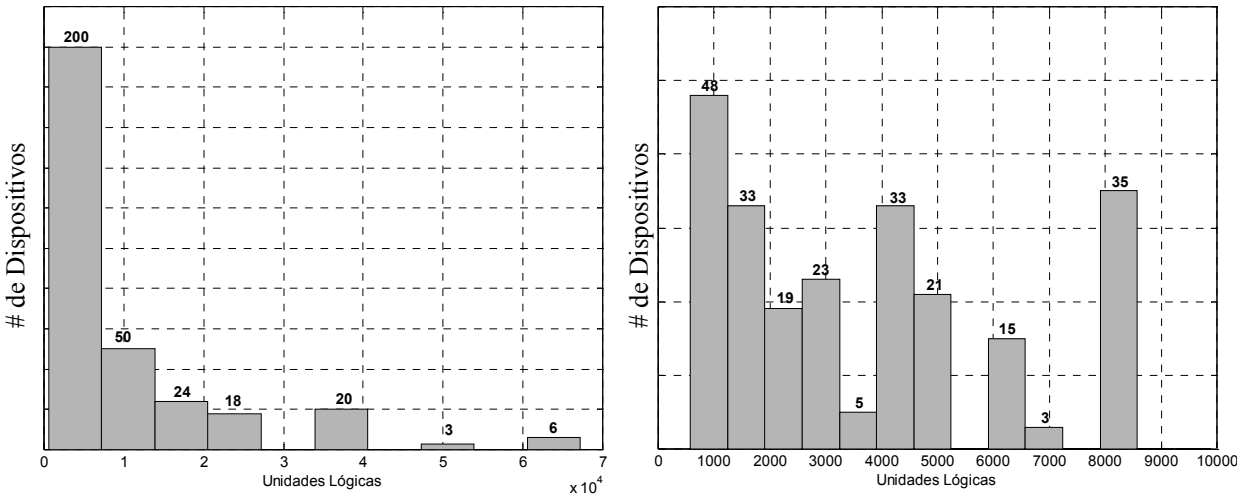
Neste capítulo são apresentados e analisados resultados obtidos nos experimentos computacionais realizados para a avaliação do algoritmo RECASTER.

7.1 Dados de Entrada

Como explicado no capítulo anterior, o RECASTER tem como entradas uma biblioteca de FPGAs e a especificação funcional e de arquitetura de um sistema embutido. Devido à ausência de *benchmarks* para a avaliação de ferramentas de síntese de sistemas embutidos, que se estende também aos sistemas embutidos com elementos redundantes, foram elaborados cinco exemplos para a avaliação do algoritmo RECASTER, quatro sintéticos e um baseado no esquema de codificação/decodificação de imagem digital JPEG. Em todos eles, foi utilizada uma mesma biblioteca de FPGAs, descrita na próxima seção.

7.1.1 Biblioteca de FPGAs

A biblioteca contém 321 FPGAs, de quatro famílias de dispositivos da Altera®, havendo 5 unidades disponíveis para cada um deles. Através de uma extensa pesquisa bibliográfica [33, 34] foi possível obter valores reais para quase todos os parâmetros dos FPGAs necessários ao RECASTER. As exceções foram os tempos de espera (Δw), as taxas de ocupação e os tempos de execução das tarefas do sistema. Esses parâmetros foram preenchidos com valores sintéticos.



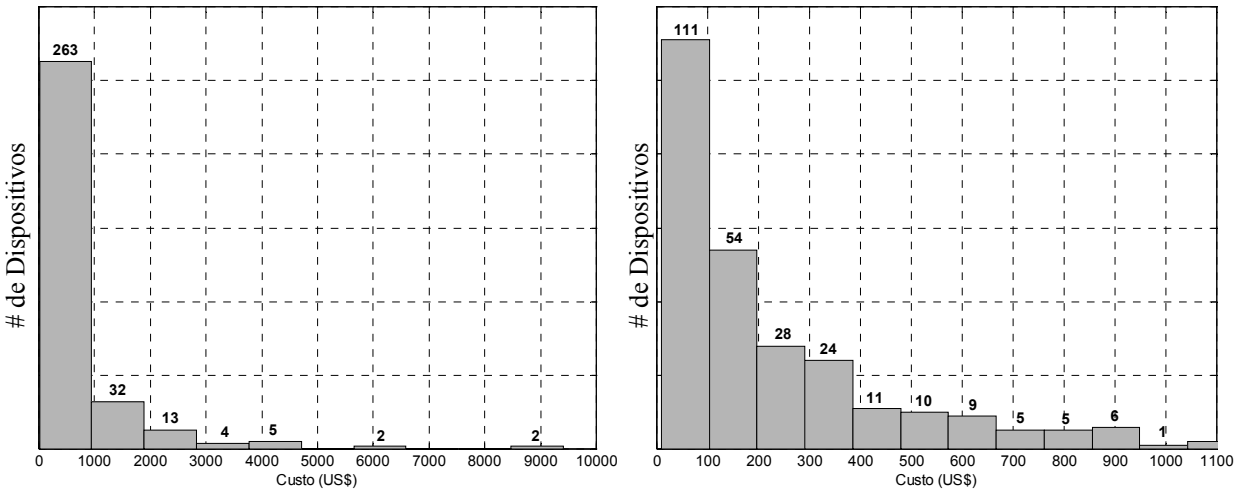
a) Biblioteca Completa b) FPGAs com # de ULs inferior à 10.000

Figura 7.1 Histogramas de Capacidade Lógica

Todos os dispositivos da biblioteca têm estruturas internas bastante semelhantes, formadas por unidades básicas denominadas LE (*Logic Element*). Isso torna a quantidade de unidades lógicas ocupadas por uma tarefa a mesma em qualquer um dos FPGAs que a implemente. Essa característica foi explorada nos exemplos sintéticos para a comparação de várias situações de ocupação de LEs pelas tarefas dos sistemas, como será mostrado mais adiante durante a descrição desses exemplos.

Capacidade Lógica

Os dispositivos de maior capacidade lógica, pertencentes à família APEXII®, possuem 67.200 LEs. Os de menor capacidade, pertencentes às famílias ACEX1K® e FLEX10K®, têm apenas 576 LEs. Para manter a nomenclatura usada no capítulo anterior, será empregado deste ponto em diante o termo Unidades Lógicas (ULs) em referência aos LEs. A diversidade da biblioteca com relação à capacidade lógica de seus componentes pode ser visualizada através de histogramas apresentando suas freqüências relativas em intervalos regulares de ULs. O histograma da Figura 7.1a contém todos os dispositivos da biblioteca e evidencia que pelo menos 200 (≈ 62% da biblioteca) têm capacidade lógica inferior à 10.000 ULs. O histograma da Figura 7.1b mostra como se dividem, em intervalos menores de ULs, esses 200 dispositivos.



a) Biblioteca Completa b) Custo < US$ 1.100,00

Figura 7.2 Histogramas de Custo

Custo

Como mostrado no histograma da Figura 7.2a, o custo dos FPGAs da biblioteca varia de maneira significativa. No entanto, percebe-se que a grande maioria dos seus componentes (263) apresenta custo inferior à US$ 1.000,00. Dentre eles, 111 têm custo inferior à US$ 100,00, como mostrado na Figura 7.2b, que apresenta um histograma mais detalhado para esses dispositivos.

Área do Encapsulamento

A área do encapsulamento é o parâmetro com menor variação relativa entre os dispositivos da biblioteca. Pelo histograma da Figura 7.3, observa-se que apenas 8 dispositivos ocupam áreas superiores à 20 cm^2. Na verdade, a maioria apresenta encapsulamentos que ocupam 12,15 ou 12,25 cm^2.

Tempo de Reconfiguração

No esquema de reconfiguração usado neste trabalho (passivo serial), definido no capítulo 5, o tempo de reconfiguração de um dispositivo pode ser aproximado por:

$$\Delta_{REC} = \frac{\text{Nº de Bits de Configuração}}{\text{Frequência Máxima de Reconfiguração}} \tag{7.1}$$

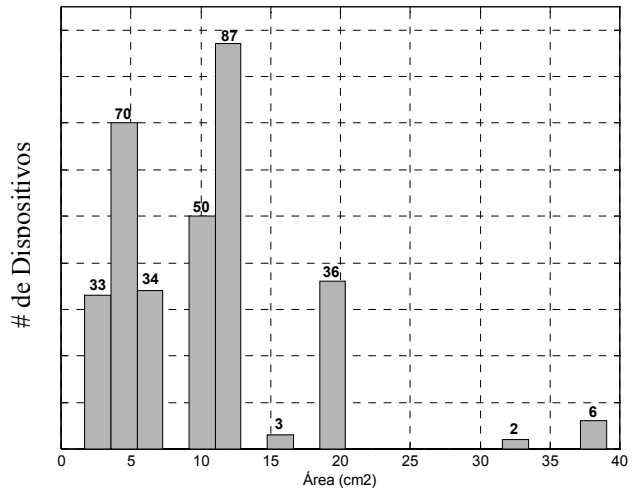
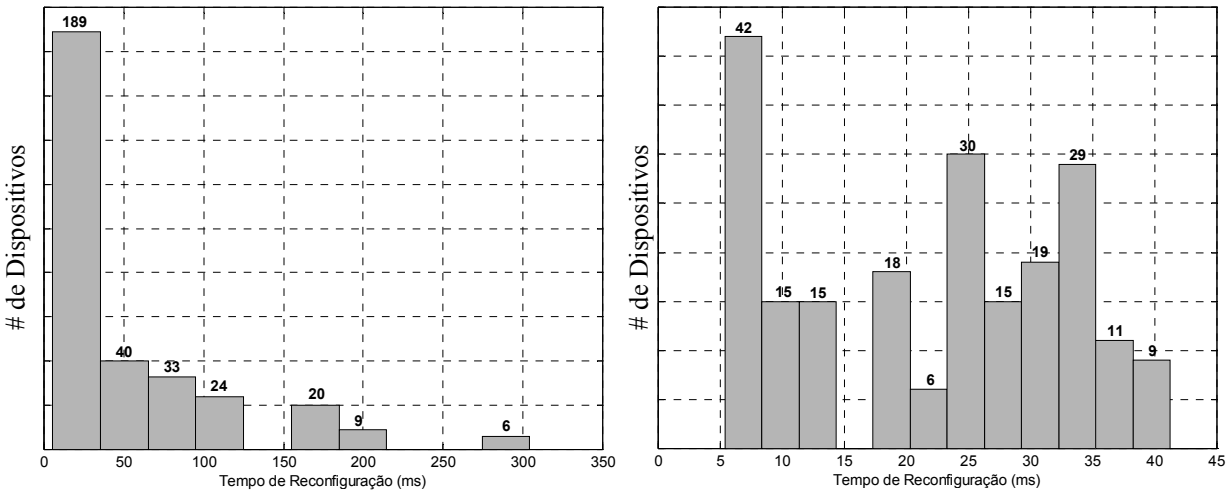


Figura 7.3 Histograma de Área

No entanto, por ter sido possível o acesso a fatores como tempos de *setup* dos dispositivos da biblioteca, a aproximação da expressão 7.1 não foi empregada, e os Δ_{REC}'s dos FPGAs foram calculados a partir de expressões disponíveis em [2], que fornecem valores bem mais realistas.

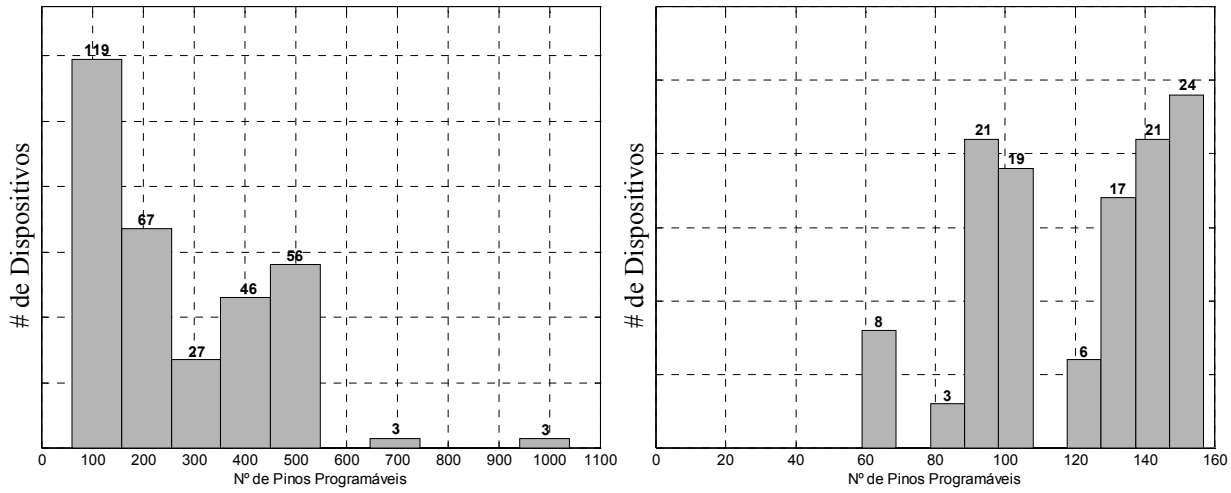
O histograma da Figura 7.4a mostra que mais de 200 dispositivos da biblioteca podem ser completamente reconfigurados em menos de 50 ms. Dentre eles, pelo menos 42 têm $\Delta_{REC} < 10$ms (Figura 7.4b).



a) Biblioteca Completa

b) $\Delta_{REC} < 45$ ms

Figura 7.4 Histogramas de Tempo de Reconfiguração



a) Biblioteca Completa

b) Nº de Pinos < 160

Figura 7.5 Histogramas de Nº de Pinos Programáveis

Os tempos de espera (Δw) para as tarefas nos dispositivos foram gerados proporcionalmente aos seus Δ_{REC}, através da relação $\Delta w = 0{,}1 \cdot \Delta_{REC}$. Isso equivale a assumir que entre duas reconfigurações de um dispositivo há, no mínimo, um intervalo de tempo correspondente a 10% de seu tempo de reconfiguração.

Número de Pinos Programáveis

O histograma da Figura 7.5a mostra como estão distribuídos os FPGAs da biblioteca com relação ao número de pinos programáveis, enquanto o da Figura 7.4b concentra-se na distribuição daqueles com menos de 160 pinos disponíveis.

Lista de Versões das Tarefas do Sistema

O tempo de execução de uma tarefa num FPGA é estimado a partir do tempo que ela gasta para executar em seu PE original, e corresponde a esse valor dividido por um *fator de desempenho*. Dispositivos de uma mesma família, com mesmo número de unidades lógicas e encapsulamento, podem apresentar desempenhos distintos na execução de uma mesma tarefa se seus *speed grades* forem diferentes. O fator de desempenho foi definido para relacionar essas

diferenças, seja entre dispositivos de uma mesma família ou não, de modo a fornecer um valor indicando o quanto um dispositivo é (ou seria) mais veloz que outro.

Os fatores de desempenho dos dispositivos da biblioteca foram obtidos pela normalização de seus desempenhos de execução pelo desempenho dos pertencentes à família ACEX1K® com *speed grade* igual a –1. Caso esses últimos executem uma certa tarefa *A* em 200 MHz, e um outro o faça em 100 MHz, seu fator de desempenho será 0,5, indicando que ele executa as tarefas do sistema em intervalos de tempo duas vezes maiores que os PEs da alocação original. Os fatores de desempenho dos FPGAs da biblioteca variam de 0,33 à 1,58.

As taxas de ocupação das tarefas são calculadas com base no número de unidades lógicas necessárias para sua implementação, definidos durante a construção do exemplo.

7.1.2 Sistemas de Entrada

Todos os exemplos sintéticos testados foram gerados através da ferramenta TGFF [19]. Desenvolvida para auxiliar na avaliação de algoritmos de síntese de sistemas embutidos, essa ferramenta é capaz de gerar, de forma pseudo-aleatória, grafos de tarefas e uma biblioteca de recursos de processamento e comunicação. A geração dos grafos é realizada a partir das seguintes informações: a) número de grafos desejado; b) número médio de nós em cada um deles; c) número máximo de arcos de entrada e saída em cada nó. Cada grafo tem um período e prazos de execução para algumas de suas tarefas.

Os componentes da biblioteca de recursos têm seus parâmetros escolhidos aleatoriamente em intervalos predeterminados. Há dois tipos de parâmetros na biblioteca: a) os associados unicamente aos dispositivos; b) os associados aos nós ou arcos dos grafos de tarefas. Tomando como exemplo um FPGA da biblioteca do RECASTER, seus parâmetros do primeiro tipo seriam: custo, área, número de pinos programáveis, capacidade lógica e quantidade disponível. Parâmetros do segundo tipo seriam as taxas de ocupação e tempos de execução das tarefas do sistema.

Nos exemplos sintéticos, os grafos de especificação funcional gerados pelo TGFF têm seus nós e arcos mapeados em PEs e *links* da biblioteca de recursos também gerada através da mesma ferramenta. A probabilidade de uma tarefa do sistema requerer redundância é definida pelo usuário do TGFF. Caso isso ocorra, PDFs são posicionados em 50 e 100% dos seus

intervalos de execução. Finalmente, cada tarefa tem atribuída uma quantidade de ULs necessárias para sua implementação na arquitetura dos FPGAs da biblioteca.

O exemplo de aplicação real foi elaborado com base em informações disponibilizadas no pacote E3S (*Embedded Systems Synthesis Benchmarks Suite*) [32], que constitui um esforço para construir e difundir *benchmarks* para o problema da síntese de sistemas embutidos. A primeira versão do E3S contém cinco conjuntos de grafos de tarefas, cada um associado à especificação funcional de um sistema embutido diferente, e uma biblioteca com 17 processadores e 6 *links* de comunicação. Sua construção foi realizada a partir de dados do EEMBC *(Embedded Microprocessor Benchmark Consortium)* [35], um consórcio de empresas que tem por objetivo disponibilizar informações sobre a implementação de tarefas relacionadas à sistemas embutidos em diversos processadores. Por enquanto, só processadores de propósito geral foram caracterizados e apenas o tempo de execução e o tamanho do código de tarefas foi disponibilizado. Os problemas encontrados na construção do "exemplo real" e as simplificações realizadas para contorná-los serão melhor discutidas no final da próxima seção, que apresenta todos os exemplos empregados na avaliação do algoritmo.

7.2 Exemplos: Descrições e Resultados Obtidos

Os resultados apresentados nesta seção foram obtidos a partir de uma versão experimental do algoritmo RECASTER, desenvolvida em linguagem C++ e executada num PC com microprocessador PentiumII®, 128MB de RAM e sistema operacional Windows98®.

7.2.1 Primeiro Exemplo Sintético

As Tabelas 7.1 e 7.2 mostram os dados para a construção do primeiro exemplo sintético através do TGFF. Na primeira tabela encontram-se os dados para a geração dos grafos. Na segunda são apresentados os valores relacionados à alocação do sistema de entrada (PEs e *Links*).

O grafo gerado é mostrado na Figura 7.6. Apenas T_6 e T_8, alocadas num único recurso de processamento e portanto tratadas como tarefas em software, não têm necessidade de redundância. Tanto elas quanto as demais executam em tempos que variam de 20 à 50 ms. Já os tempos das comunicações variam de 9 à 11 ms.

# de Grafos	1
# Médio de Nós	10
# Max. de Arcos de Entrada	2
# Max. de Arcos de Saída	2

Tabela 7.1 Geração de Grafos

			Valor Médio	Variação
Nós (Tarefas)	PEs	Tempos de Execução (UT)	35	15
	# de Pinos de Interface		36	24
Arcos	*Links*	Tempos de Comunicação (UT)	10	1

Tabela 7.2 Geração de Alocação de Entrada

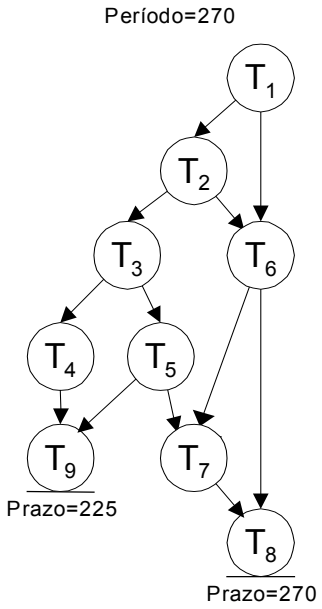


Figura 7.6 Grafo (Exemplo 1)

A variação no número de pinos de interface das tarefas foi escolhida de modo que todos os FPGAs da biblioteca possuíssem pinos suficientes para a implementação isolada de qualquer tarefa do sistema. Com isso, o número de ULs ocupadas pelas tarefas tornou-se a principal restrição durante a síntese de redundância coletiva e seus efeitos sobre esse procedimento puderam ser melhor avaliados.

O algoritmo foi executado para quatro faixas de variação do número médio de ULs ocupados pelas tarefas do sistema, nos dois critérios de otimização possíveis. A Tabela 7.3 mostra os resultados obtidos buscando-se uma alocação de menor custo, enquanto na Tabela 7.4 são apresentados os resultados buscando-se uma alocação de menor área.

A primeira faixa de variação do número de ULs foi escolhida de modo que todos os FPGAs da biblioteca tivessem capacidade lógica suficiente para implementar, isoladamente, qualquer tarefa do sistema. Seu valor médio (476) somado à sua máxima variação positiva (100) corresponde ao número de ULs do FPGA de menor capacidade lógica da biblioteca. Os valores médios e as variações das faixas seguintes são múltiplos dessa primeira (5x ,10x e 20x).

Nota-se pelas Tabelas 7.3 e 7.4 que o aumento do número médio de ULs por tarefa provoca o aumento do tempo de execução do algoritmo. Isso indica que a ordenação das alocações, seja por custo ou área, imprime também uma ordenação, indireta, de capacidade lógi-

Critério de Otimização			Custo			
# de ULs (Médio ± Variação)			476 ± 100	2.380 ± 500	4.760 ± 1.000	9.520 ± 2.000
Tempo de Execução (s)			5,5	5,6	5,7	5,8
Solução Encontrada	Alocação	FPGA	EP1K30TC1443	EP1K50TC1443	EP20K160EQC2083	EP20K300EQC2403
		Fator de Desempenho	0,6491	0,6491	0,7497	0,7497
		Quantidade	1	2	2	3
		Custo (US$)	14,00	40,00	308,00	768,00
		Área (cm^2)	4,84	9,68	18,42	36,45
	Prazos	Violação Média (%)	11,0	11,0	12,7	14,0
		Violação Máxima (%)	16,6	16,6	18,6	19,3

Tabela 7.3 Exemplo 1: Otimização do Custo da Alocação

Critério de Otimização			Área			
# de ULs (Médio ± Variação)			476 ± 100	2.380 ± 500	4.760 ± 1.000	9.520 ± 2.000
Tempo de Execução (s)			5,7	5,8	6,2	6,4
Solução Encontrada	Alocação	FPGA	EP1K50FC2563	EP20K60ETC1441	EP20K160ETC1442	EP20K300EFC6723
		Fator de Desempenho	0,6491	1,2231	0,9956	0,7497
		Quantidade	1	2	2	3
		Custo (US$)	28,00	112,00	328,00	1.008,00
		Área (cm^2)	2,89	3,38	9,68	21,87
	Prazos	Violação Média (%)	11,3	12,6	16,4	14,0
		Violação Máxima (%)	16,6	18,2	23,4	19,3

Tabela 7.4 Exemplo 1: Otimização da Área da Alocação

ca. Quanto maior a demanda por ULs, mais alocações serão testadas até que se encontre uma válida para o sistema, daí o aumento no tempo de execução do algoritmo. Embora essa seja uma tendência observada em todos os exemplos sintéticos, é importante lembrar que o RECASTER não trabalha apenas com restrições de recursos lógicos na síntese de redundância coletiva. Devem também ser respeitadas restrições de tempo real na determinação dos escalonamentos dos cenários de falha requeridos para o sistema. Se isso não ocorresse, as alocações determinadas para as terceiras faixas de variação de número de ULs, em ambos critérios de otimização, seriam provavelmente formadas por alocações com o dobro de FPGAs das alocações empregadas na

faixa anterior. Isso porque, além de estarem presentes na lista de alocações possíveis para o sistema, essas alocações teriam recursos lógicos suficientes e custo e área (respectivamente nas Tabelas 7.3 e 7.4) menores que as soluções determinadas pelo algoritmo. No caso da síntese com otimização de custo, uma alocação com 4 FPGAs EP1K50TC1443, o dobro da quantidade utilizada na segunda faixa de variação de ULs, custaria US$ 80,00, um valor bem inferior aos US$ 308,00 da solução definida pelo algoritmo.

Entre todas as alocações definidas para as situações avaliadas, apenas uma é composta por dispositivos com fator de desempenho maior que 1, ou seja, FPGAs que executam as tarefas do sistema mais rápido que seus PEs. Nas demais alocações predominam FPGAs em média 25% mais lentos que eles. Isso ocorre especialmente em todas as alocações obtidas com critério de otimização de custo, indicando que o algoritmo faz uso da margem de tempo do sistema para contornar o desempenho inferior dos FPGAs e obter escalonamentos válidos para os cenários de falhas simples requeridos.

Os atrasos impostos às tarefas do sistema durante a obtenção do escalonamento do cenário de ausência de falhas (∅) têm grande influência na determinação das violações de prazos no caso de falha e re-execução dessas tarefas. Nas Figuras 7.7a-7.7d são mostrados os escalonamentos obtidos pelo algoritmo para os cenários ∅ em todas as faixas de variação de ULs testadas com critério de otimização de custo. Neles é possível observar o aumento dos tempos de reconfiguração dos FPGAs à medida que cresce o número médio de ULs requeridos pelas tarefas do sistema. Na tentativa de manter uma alocação com um número mínimo de FPGAs, as tarefas sofrem maiores atrasos, o que reflete no aumento das violações média e máxima em caso de falhas no sistema, observada na Tabela 7.3. Alguns desses atrasos podem ser visualizados através das Figuras 7.7a-7.7d (tarefas mapeadas nos PEs 6 e 8). O aumento no número de FPGAs das alocações decorre da incapacidade de serem cumpridos os prazos do sistema apenas pelo atraso da execução de suas tarefas.

Redundância Coletiva *x* Redundância Convencional

Caso fosse empregada no sistema uma estratégia convencional de redundância de hardware, com um PE reserva para a execução de cada tarefa tolerante a falhas, seria necessária uma quantidade duas vezes maior de PEs em hardware no projeto final. Neste exemplo isso corresponderia à um acréscimo de 7 PEs. Assumindo que todos têm encapsulamentos ocupando

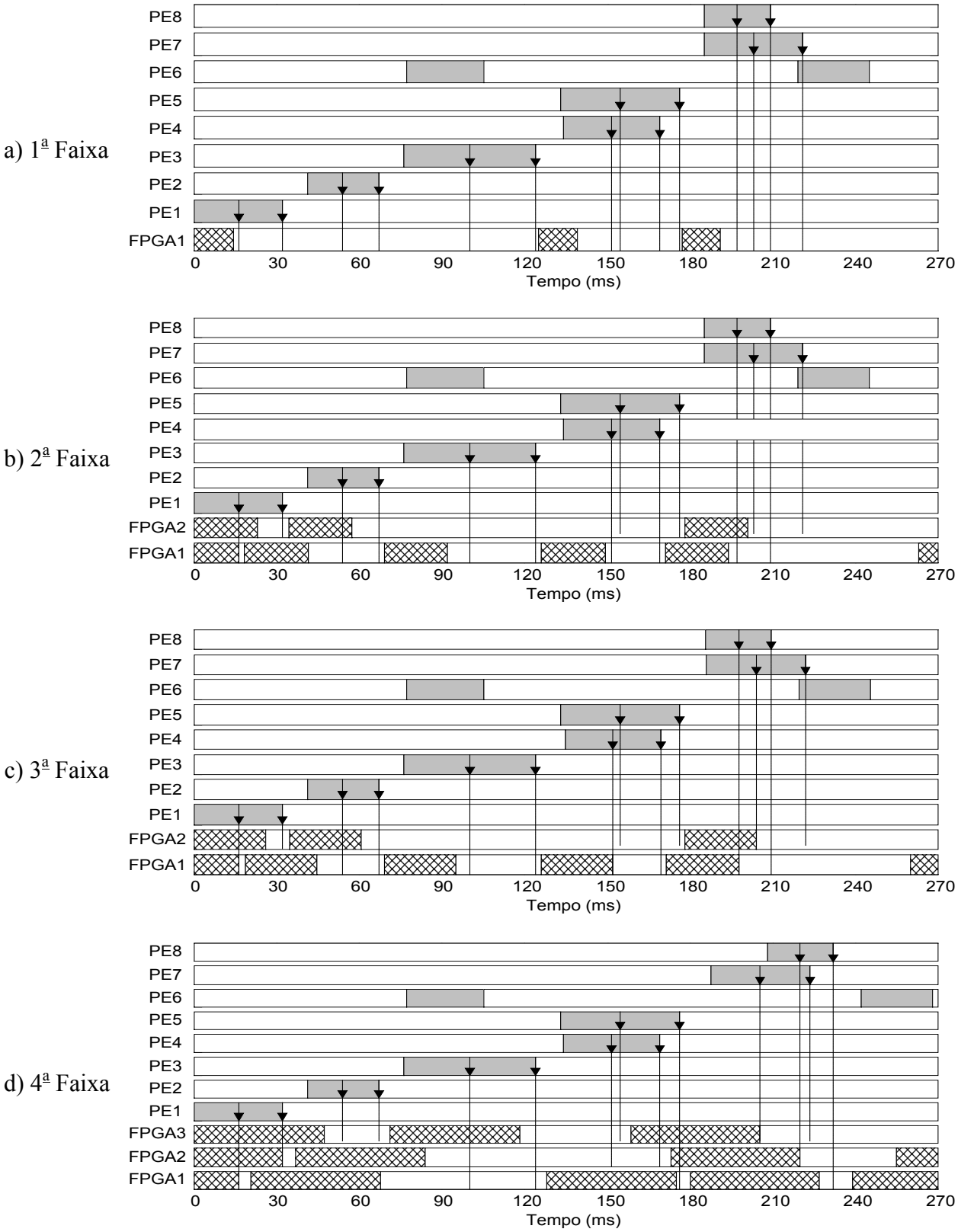


Figura 7.7 Escalonamentos dos Cenários ∅ nas Faixas de Variação de ULs (Custo)

uma mesma área, para que a soma dessas áreas seja menor que a área ocupada pela redundância coletiva determinada pelo algoritmo, cada um deles deveria ocupar 1/7 da área da alocação de FPGAs do sistema. Para a primeira situação de número de ULs por tarefa, com critério de otimização de custo, isso corresponderia à 0,6914 cm^2, que equivale à uma área aproximadamente 2,5 vezes inferior à ocupada pelo FPGA de menor encapsulamento na biblioteca. Tomando a alocação obtida com otimização de área, essa relação seria de aproximadamente 4 vezes. Considerando factível a implementação das tarefas através de dispositivos com tais dimensões, o fato de encapsulamentos menores serem, em geral, mais caros, implica na necessidade de serem analisados também os custos de ambas implementações, de modo que seja escolhida a estratégia de melhor relação custo-área para o projeto.

É importante lembrar que o emprego de redundância coletiva implica na necessidade de memória adicional para armazenar *bitstreams* de programação dos FPGAs, além de possivelmente mecanismos de roteamento programáveis para os pinos de saída desses dispositivos. O aumento de área e custo causado por esses elementos também devem ser considerados numa análise mais precisa das opções de projeto possíveis. Neste trabalho, no entanto, a comparação entre a estratégia convencional de duplicação de PEs e a de redundância coletiva será limitada à definição da área média dos PEs redundantes de modo que no total ocupem o mesmo que a alocação de FPGAs definida pelo RECASTER. Apenas as alocações das primeiras faixas de variação de número de ULs por tarefa serão consideradas para isso. O aumento do número de ULs requerido por tarefa certamente implicaria na necessidade de áreas de hardware maiores para sua implementação através de hardware dedicado. Optamos no entanto por evitar tais estimativas de modo à não incorrer em erros por falta de informações sobre esse tema.

7.2.2 Segundo Exemplo Sintético

O segundo exemplo sintético foi gerado com os dados das Tabelas 7.5 e 7.6. Essa última apresenta os mesmos valores mostrados na Tabela 7.2, indicando que os PEs e *Links* empregados nesse exemplo têm características semelhantes aos utilizados no anterior. Há porém grandes diferenças entre as especificações funcionais dos dois exemplos. Para o segundo foi gerado um sistema multi-taxa, especificado através de dois grafos, um com 9 tarefas e outro com 18. Além

# de Grafos	2
# Médio de Nós	10
# Max. de Arcos de Entrada	3
# Max. de Arcos de Saída	3

Tabela 7.5 Geração de Grafos

			Valor Médio	Variação
Nós (Tarefas)	PEs	Tempos de Execução (UT)	35	15
		# de Pinos de Interface	36	24
Arcos	*Links*	Tempos de Comunicação (UT)	10	1

Tabela 7.6 Geração de Alocação de Entrada

disso, suas dependências de comunicação são mais complexas, pela maior quantidade de arcos de entrada e saída possíveis em cada uma de suas tarefas. Devido à complexidade desses grafos, suas representações gráficas são mostradas apenas no Apêndice A, tal como fornecidas pelo TGFF.

O período de execução do menor grafo é 250 ms. Ele deve executar duas vezes ao longo do hiperperíodo do sistema, que corresponde ao período do maior grafo, 500 ms. Há portanto 36 execuções de tarefas para serem escalonadas por cenário de falhas, das quais 24 correspondem às tarefas com necessidade de redundância, cada uma com 2 PDFs para serem cobertos durante o escalonamento do cenário ∅.

Assim como no exemplo anterior, o algoritmo foi executado para quatro faixas de variação do número médio de ULs ocupadas por tarefa do sistema, nos dois critérios de otimização possíveis. Essas faixas correspondem às mesmas empregadas no primeiro exemplo, e os resultados obtidos para cada uma delas são mostrados nas Tabelas 7.7 e 7.8. Foram encontradas soluções apenas para as 3 primeiras faixas de variação de ULs. Na última, todas as alocações da lista foram testadas sem sucesso, daí o mesmo tempo de execução do algoritmo nos dois critérios de otimização.

Tal como no primeiro exemplo, nota-se pelos dados fornecidos pelo RECASTER que a otimização de área implica freqüentemente em alocações de FPGAs de custo bem mais elevado que as obtidas buscando-se alternativas de menor custo. Podem no entanto ser encontradas soluções com boa relação entre esses dois parâmetros, bastando para isso que se compare as soluções fornecidas pelo algoritmo nos dois critérios de otimização. A alocação obtida na terceira faixa de variação de ULs, com critério de otimização de custo, custa US$ 770,00 e ocupa uma área de 46,06 cm^2. Com critério de otimização de área obtém-se uma alocação de custo US$ 820,00 e área 24,02 cm^2. Esses dados mostram que a alocação de menor área, apesar de custar

Critério de Otimização			Custo			
# de ULs (Médio ± Variação)			476 ± 100	2.380 ± 500	4.760 ± 1.000	9.520 ± 2.000
Tempo de Execução (s)			11,9	13,1	16,7	29,4
Solução Encontrada	Alocação	FPGA	EP1K10TC1003	EP1K50TC1443	EP20K160EQC2083	-
		Fator de Desempenho	0,6491	0,6491	0,7497	-
		Quantidade	3	5	5	-
		Custo (US$)	24,00	100,00	770,00	-
		Área (cm^2)	7,86	24,2	46,05	-
	Prazos	Violação Média (%)	5,9	8,2	8,5	-
		Violação Máxima (%)	41,1	31,1	35,9	-
	Média das Taxas de Ocupação dos FPGAs (%)		82,5	82,8	73,8	-

Tabela 7.7 Exemplo 2: Otimização de Custo da Alocação

Critério de Otimização			Área			
# de ULs (Médio ± Variação)			476 ± 100	2.380 ± 500	4.760 ± 1.000	9.520 ± 2.000
Tempo de Execução (s)			12,0	15,9	22,0	29,4
Solução Encontrada	Alocação	FPGA	EP20K30EFC1443	EP20K100EFC1443	EP20K160ETC1442	-
		Fator de Desempenho	0,7497	0,7497	0,9956	-
		Quantidade	2	5	5	-
		Custo (US$)	64,00	350,00	820,00	-
		Área (cm^2)	3,380	8,45	24, 2	-
	Prazos	Violação Média (%)	6,5	9,4	12,2	-
		Violação Máxima (%)	39,4	35,9	47,7	-
	Média das Taxas de Ocupação dos FPGAs (%)		77,6	60,6	73,8	-

Tabela 7.8 Exemplo 2: Otimização de Área da Alocação

apenas 6,5% a mais que a de menor custo, ocupa uma área 47,44% menor. Desde que o aumento nas violações seja tolerável, esta é provavelmente a melhor solução entre as duas.

Percebe-se nas Tabelas 7.7 e 7.8 uma diferença significativa entre as violações percentuais médias e máximas em todas as variações do exemplo. Isso ocorre porque em sistemas

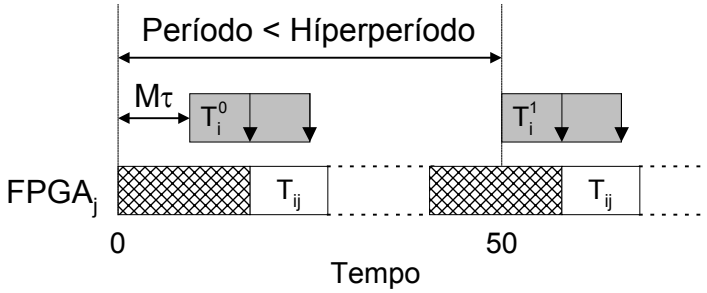


Figura 7.8 Motivo para Violação Média << Violação Máxima

multi-taxa, pelo fato de alguns grafos executarem mais de uma vez, FPGAs podem ter suas reconfigurações iniciadas entre duas execuções consecutivas. Isso permite que eles estejam prontos para cobrir os PDFs das execuções posteriores mais cedo, geralmente evitando ou reduzindo a necessidade de atrasá-los. A Figura 7.8 ilustra isso, como comentado a seguir.

Suponha que um determinado grafo com período P=50 UT execute várias vezes ao longo do hiperperíodo de um sistema qualquer. Além disso, que uma de suas tarefas, T_i, tenha suas execuções no escalonamento *EARLY* iniciadas em Nc·P, ou seja, a primeira em 0 UT, a segunda em 50 UT e assim sucessivamente. Admita ainda que, para que o primeiro PDF de T_i^0 seja coberto através de um FPGA$_j$, sua execução seja atrasada em Mτ UT, possivelmente aumentando a violação de prazos decorrente de falhas na primeira execução do grafo, e que na seqüência sejam escalonadas reconfigurações e tarefas redundantes em FPGA$_j$, até que o primeiro PDF da segunda execução de T_i torne-se o *PDF corrente*. Tendo sido respeitados todos os tempos de espera para os PDFs pertencentes à primeira execução do grafo, a reconfiguração de FPGA$_j$ para cobrir p_{i0}^1, se necessária, pode iniciar antes mesmo de 50 UT, evitando que T_i^1 precise ser atrasada. A repetição dessa situação para diversas tarefas do sistema, tende a diminuir a violação média de prazos com relação à violação máxima, que geralmente acaba sendo definida por falhas ainda nas primeiras execuções dos grafos.

Nas Tabelas 7.7 e 7.8 são acrescentadas as médias das taxas de ocupação dos FPGAs das alocações, uma informação que indica o número de unidades lógicas que permanecem, em média, ociosas durante os escalonamentos ∅ das variações do exemplo. Essa é uma das informações possíveis de serem extraídas dos arquivos de escalonamentos de FPGAs fornecidos pelo RECAS-TER, que têm taxas de ocupação e número de pinos utilizados associados à cada configuração.

Com o objetivo de avaliar todo o conjunto de soluções para o sistema na terceira faixa de variação de ULs, a condição de parada do algoritmo (determinação de uma solução válida) foi relaxada. As soluções obtidas são mostradas na Figura 7.9, plotadas num espaço de coordenadas custo, área e violação percentual máxima. A solução de menor custo é representada por um quadrado e a de menor área por um círculo.

Embora eficaz para se visualizar a distribuição das alocações no espaço de projeto da redundância coletiva do sistema, a Figura 7.9 deve ser avaliada em conjunto com gráficos bidimensionais, como os apresentados nas Figuras 7.10 e 7.11. Neles percebe-se melhor as diferenças existentes entre os parâmetros custo e área das alocações.

Apesar de uma versão do RECASTER ter sido construída com esse intuito, não é sempre que a exploração de todas as soluções de redundância coletiva de um sistema é computacionalmente viável. A complexidade do sistema, o número de cenários de falha possíveis e o tamanho da lista de alocações de FPGAs são os principais fatores na determinação do tempo de execução do algoritmo e sua aplicação deve ser realizada levando-se isso em consideração. Na

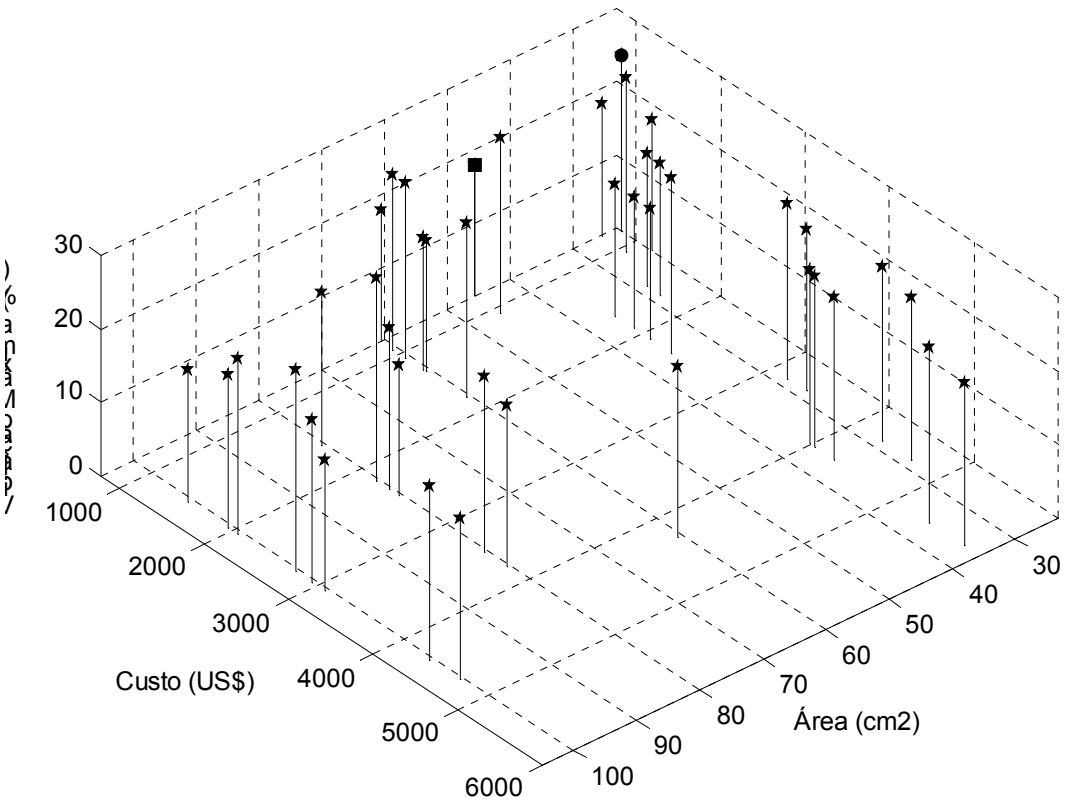


Figura 7.9 Exemplo 2: Distribuição das Soluções na 3ª Faixa de Variação de ULs

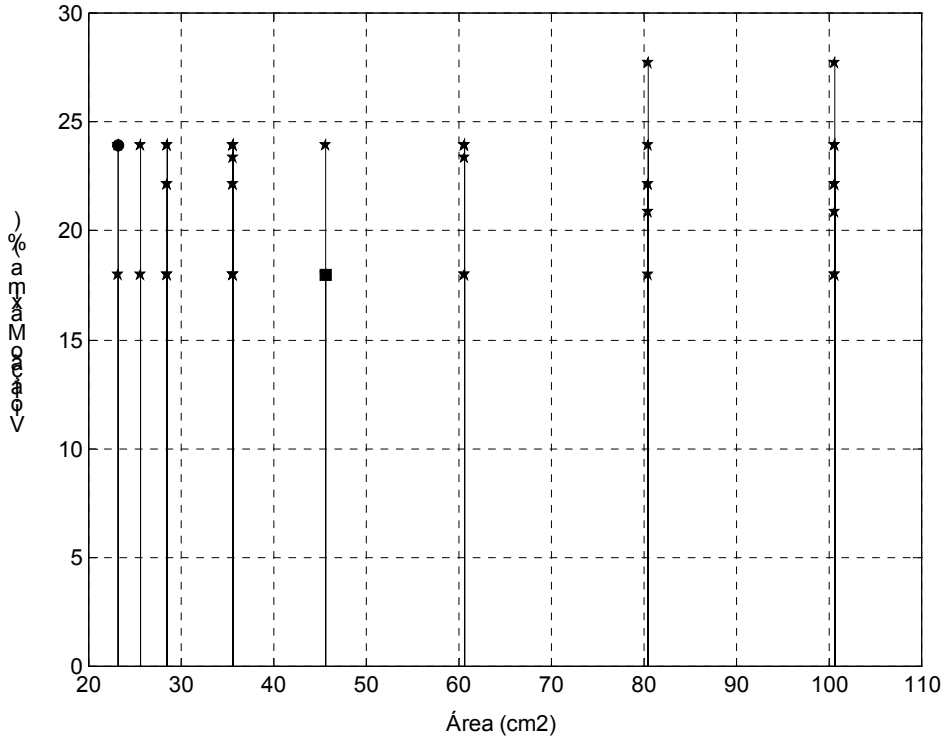


Figura 7.10 Exemplo 2: *Violação Máxima x Área* para Todas as Soluções

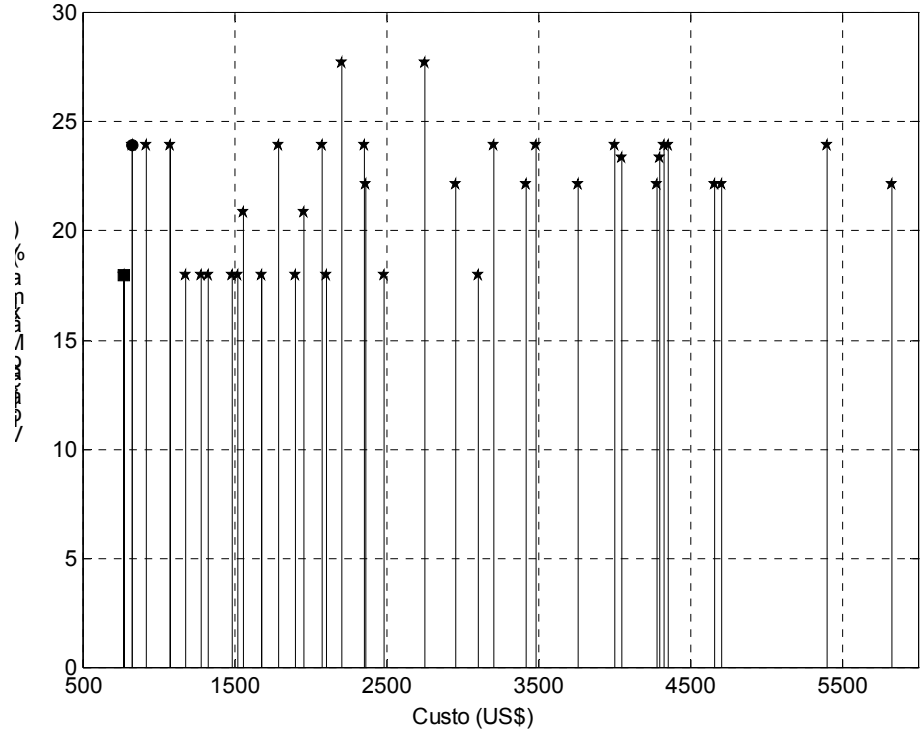


Figura 7.11 Exemplo 2: *Violação Máxima x Custo* para Todas as Soluções

seção 7.2.4 será abordado novamente o problema dos tempos de execução do algoritmo.

Redundância Coletiva *x* Redundância Convencional

Neste exemplo, caso fossem empregados PEs reserva para cada tarefa com necessidade de redundância, a área total ocupada por eles seria equivalente às alocações de FPGAs definidas pelo algoritmo apenas se ocupassem, em média, 0,4367 cm^2 e 0,1878 cm^2, considerando os resultados obtidos com critérios de otimização de custo e área, respectivamente, nas primeiras faixas de variação de ULs por tarefa. O primeiro valor seria 3,9 vezes inferior à área do menor encapsulamento de FPGA da biblioteca, enquanto o segundo seria 9 vezes inferior à mesma.

7.2.3 Terceiro Exemplo Sintético

Com o objetivo de avaliar o algoritmo para sistemas com um número de tarefas bem mais elevado que nos exemplos anteriores, foram utilizados como entrada para o TGFF os dados mostrados nas Tabelas 7.9 e 7.10. Essa última, assim como no segundo exemplo, permanece inalterada.

# de Grafos	2
# Médio de Nós	30
# Max. de Arcos de Entrada	2
# Max. de Arcos de Saída	2

Tabela 7.9 Geração de Grafos

			Valor Médio	Variação
Nós (Tarefas)	PEs	Tempos de Execução (UT)	35	15
		# de Pinos de Interface	36	24
Arcos	*Links*	Tempos de Comunicação (UT)	10	1

Tabela 7.10 Geração de Alocação de Entrada

Foi gerado um sistema multi-taxa com hiperperíodo igual à 2.090 ms. Sua especificação funcional é composta por dois grafos, um com 60 tarefas e período de execução 1.045 ms e outro com 90 tarefas e período de execução 2.090 ms. Devem portanto ser escalonadas 210 execuções de tarefas por cenário de falhas, das quais 83 correspondem às tarefas com necessidade de redundância. Há ao todo 64 cenários de falhas para serem explorados. Os grafos gerados são mostrados na seção A.1.3 do apêndice A.

Os testes realizados foram os mesmos dos exemplos anteriores: otimização de custo e área em quatro variações do número médio de ULs ocupado por tarefa do sistema. Os dados obtidos são mostrados nas Tabelas 7.11 e 7.12. Diferente do que ocorreu no segundo exemplo, foram encontradas soluções em todas essas variações, evidenciando que a complexidade do sistema de entrada não é o único fator para o sucesso na síntese de redundância do sistema.

Critério de Otimização			Custo			
# de ULs (Médio ± Variação)			476 ± 100	2.380 ± 500	4.760 ± 1.000	9.520 ± 2.000
Tempo de Execução (s)			205,3	640,8	809,1	724,2
Solução Encontrada	Alocação	FPGA	EP1K10TC1003	EP20K100ETC1443	EP20K400EFC6723	EP20K1500EBC6523
		Fator de Desempenho	0,6491	0,7497	0,7497	0,7497
		Quantidade	3	5	5	5
		Custo (US$)	24,00	345,00	1.900,00	6.625,00
		Área (cm^2)	7,86	24,20	36,45	101, 25
	Prazos	Violação Média (%)	0,0	0,04	0,22	0,30
		Violação Máxima (%)	0,0	1,45	3,76	4,18
	Média das Taxas de Ocupação dos FPGAs (%)		79,99	63,04	82,24	80,42

Tabela 7.11 Exemplo 3: Otimização de Custo da Alocação

Critério de Otimização			Área			
# de ULs (Médio ± Variação)			476 ± 100	2.380 ± 500	4.760 ± 1.000	9.520 ± 2.000
Tempo de Execução (s)			223,2	362,6	740,6	773,1
Solução Encontrada	Alocação	FPGA	EP20K30EFC1443	EP20K100EFC1443	EP20K1000EFC6723	EP2A70B724C9
		Fator de Desempenho	0,7497	0,7497	0,7497	0,9720
		Quantidade	2	5	4	5
		Custo (US$)	64,00	350,00	2.600,00	23.000,00
		Área (cm^2)	3,38	8,45	29,16	61,25
	Prazos	Violação Média (%)	0,0	0,04	0,12	0,66
		Violação Máxima (%)	0,0	1,45	6,05	9,19
	Média das Taxas de Ocupação dos FPGAs (%)		71,67	63,04	79,23	80,38

Tabela 7.12 Exemplo 3: Otimização de Área da Alocação

Ocorre entre as alocações obtidas nos dois critérios de otimização, na segunda faixa de variação de ULs, situação semelhante à observada no segundo exemplo entre as alocações da terceira faixa. Aqui porém é obtida uma redução de área bem maior que no exemplo anterior, através de um aumento de custo inferior. Um aumento de 1,45% no custo reduz em 65,08% a área da alocação, sem alterações nas violações de prazo do sistema.

Na Tabela 7.12 um dado que se destaca é o elevado custo da alocação obtida na quarta faixa de variação de ULs. Essa alocação é composta por dispositivos pertencentes à um dos cinco tipos de FPGAs da biblioteca que apresentam custos em torno de US$ 4.500,00, como pode ser observado no histograma da Figura 7.2a. A solução de menor custo nessa faixa é 71,19% mais barata que ela, porém tem área 65,3% maior. Pela proximidade entre esses valores, é difícil definir qual seria a melhor para o sistema. Em casos como esse, uma maior exploração do espaço de projeto, como mostrada no exemplo anterior, poderia auxiliar o projetista nessa tarefa.

Redundância Coletiva *x* Redundância Convencional

O emprego de PEs reserva seria equivalente (em área ocupada) às alocações de FPGAs obtidas pelo algoritmo apenas se suas áreas fossem, em média, de 0,1248 cm^2 (custo) e 0,0537 cm^2 (área). Tais encapsulamentos provavelmente não poderiam conter a quantidade média de pinos requeridos pelas tarefas do sistema, inviabilizando o projeto.

7.2.4 Quarto Exemplo Sintético: Tempos de Execução do Algoritmo

Pode-se perceber ao longo dos exemplos anteriores uma variação significativa nos tempos de execução do algoritmo. Como dito anteriormente, esses tempos dependem fortemente da complexidade do sistema e do número de cenários de falhas que devem ser explorados para sua versão tolerante a falhas. Uma vez que a complexidade dos sistemas nos três exemplos varia tanto no número de tarefas quanto na topologia dos grafos, os tempos gastos na obtenção de suas soluções não são adequados para avaliar genericamente o tempo de execução do algoritmo. Com esse objetivo, foram gerados através do TGFF diversos sistemas, de um grafo apenas, com várias quantidades distintas de tarefas. Nesses sistemas, todas as tarefas têm necessidade de redundância, tornando o número de cenários de falhas igual ao total de tarefas mais uma unidade.

				Tempo de Execução (s)													
# de Tarefas				10	20	30	40	50	60	70	80	90	100	125	150	175	200
Otimização	Custo	Fase	1	5,51	8,38	11,51	13,97	16,58	19,72	22,30	25,56	28,11	30,72	38,40	46,70	55,22	62,43
			2&3	0,08	0,54	1,92	3,07	4,82	6,75	9,42	16,96	25,52	31,47	51,72	95,03	149,88	221,23
	Área	Fase	1	5,41	8,13	11,17	13,94	16,45	19,56	22,14	25,52	27,88	31,03	39,06	46,07	54,82	62,16
			2&3	0,09	0,32	1,51	2,39	7,42	9,67	13,02	18,71	22,56	34,01	69,61	92,00	213,66	245,40

Tabela 7.13 Tempos de Execução

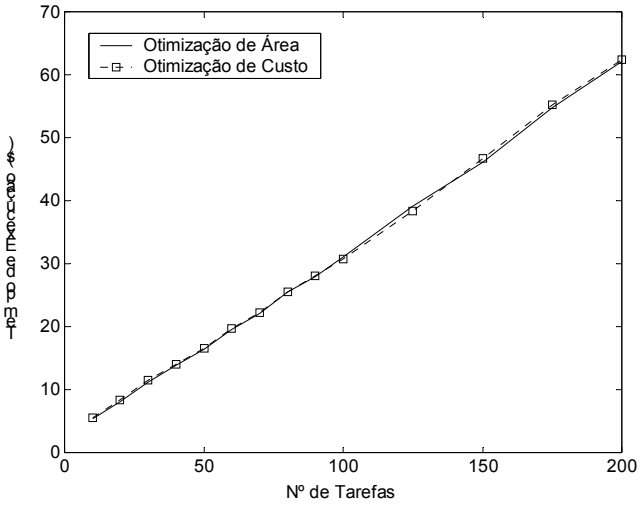


Figura 7.12 Tempos de Execução: Fase 1

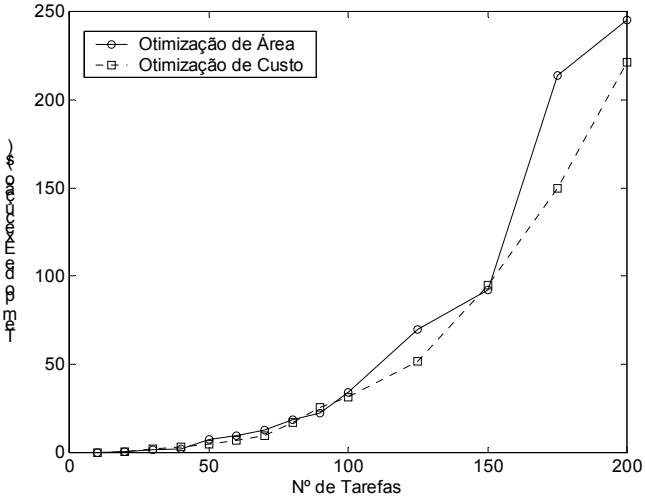


Figura 7.13 Tempos de Execução: Fases 2 e 3

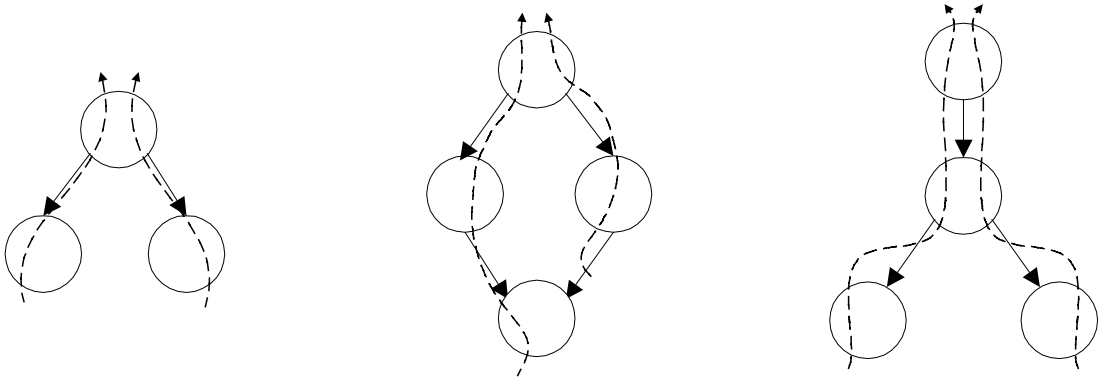
Foram gerados ao todo 14 grafos (sistemas) e anotados os tempos de execução do RECASTER tendo-os como entrada, nos dois critérios de otimização possíveis. Esses tempos foram tomados distinguindo-se a parcela gasta durante a execução das etapas da fase 1, da parcela referente à execução das etapas das fases 2 e 3. Os números de tarefas e os tempos obtidos são mostrados na Tabela 7.13. Através dos seus dados, foram plotados os gráficos das Figuras 7.12 e 7.13, que apresentam a evolução dos tempos de execução do algoritmo com o aumento do número de tarefas do sistema de entrada.

Percebe-se pela Figura 7.12 uma variação linear no tempo gasto durante a execução da primeira fase do algoritmo com relação ao tamanho do grafo de entrada, um comportamento esperado pelo fato de haver uma quantidade fixa de informações para serem lidas por tarefa do sistema. Já para as fases 2 e 3 percebe-se uma relação não-linear entre os tempos de execução de suas etapas e o tamanho do grafo de entrada (Figura 7.13). No entanto, tal comportamento e especialmente a forma como ocorre neste exemplo, não pode ser generalizado para outras entradas do RECASTER.

Há diversos fatores que impedem que se encontre uma relação entre o número de tarefas de um grafo de entrada para o algoritmo e seu tempo de execução. A primeira, e também mais óbvia, é a possibilidade de não ser encontrada uma solução, mesmo sem restringir a quantidade de FPGAs numa alocação. Ainda que as tarefas de um sistema tenham FPGAs exclusivos como redundância, permanece a restrição de que suas execuções nesses dispositivos devem permitir a determinação de escalonamentos válidos para seus cenários de falhas correspondentes. Se essa restrição for relaxada para que tais escalonamentos sempre sejam encontrados, igualando à 1 os fatores de desempenho dos FPGAs da biblioteca, a redundância coletiva equivalerá à redundância em hardware dedicado (seção 5.1.2), e o uso de hardware reconfigurável terá perdido o sentido.

Outra característica do RECASTER que impede estimar seu tempo de execução com base no número de tarefas do sistema de entrada é a relação existente entre a topologia dos seus grafos e o tempo gasto no procedimento de determinação de margens iniciais. O procedimento de determinação dos escalonamentos ALAP das tarefas exemplifica claramente o problema. Considere que devam ser determinados esses escalonamentos para as tarefas dos grafos da Figura 7.14. Em cada um deles são mostrados através de setas tracejadas os percursos que devem ser realizados a partir de suas tarefas de saída. A quantidade de vezes que uma tarefa é atravessada por uma dessas setas corresponde ao número de atualizações sofrido por seu escalonamento ALAP. Percebe-se pela Figura 7.14 que o mínimo acréscimo no número de tarefas num grafo, isto é, a adição de uma tarefa, pode resultar em grafos com topologias e tempos gastos em procedimentos de escalonamento ALAP distintos, devido aos diferentes números de atualizações realizadas.

As duas variações do grafo da Figura 7.14a têm a mesma *profundidade*, isto é, a mesma quantidade máxima de nós ao longo de um ramo. Caso a tarefa acrescida ao grafo fosse dependente apenas de sua tarefa de entrada, sua profundidade teria sido mantida. Os grafos



a) 4 Atualizações b) 5 Atualizações c) 6 Atualizações

Figura 7.14 Relação Topologia ↔ ALAP

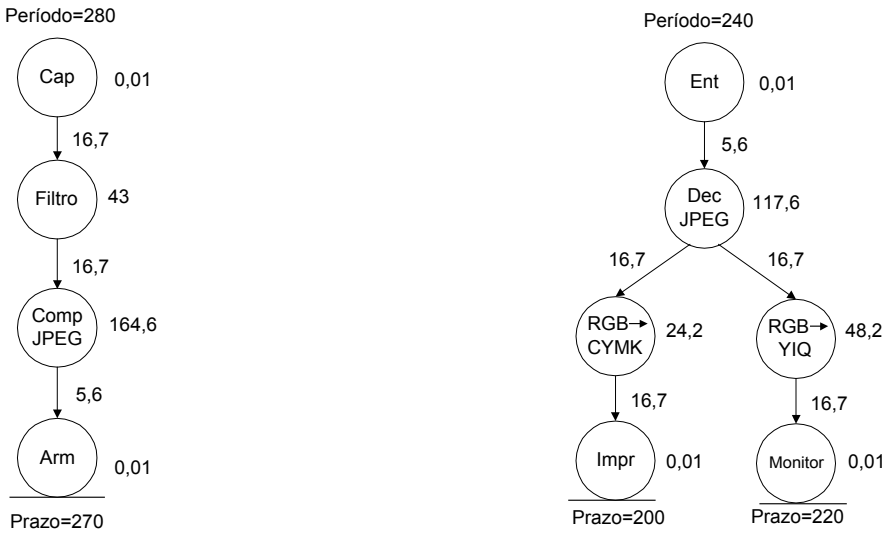
empregados na obtenção dos valores da Tabela 7.13 foram gerados pelo TGFF com profundidades distintas e sem relação com suas quantidades de tarefas. Isso porque os números máximo e mínimo de arcos de entrada e saída por tarefa foram fixados em 2. Mesmo que esses valores fossem fixados em 1, não haveria garantias de uma variação regular das profundidades dos grafos com o aumento do número de tarefas. Essa liberdade na geração dos grafos pode ser observada na seção A.1.4 do apêndice A, sendo um dos motivos pelo qual a curva da Figura 7.13 deve ser tratada como válida apenas para o exemplo desta seção.

7.2.5 Exemplo de uma Aplicação Real

A Figura 7.15 mostra a especificação funcional de um sistema embutido capaz de: a) receber uma imagem digital no espaço de cores RGB e codificá-la para o formato JPEG; b) receber uma imagem no formato JPEG e decodificá-la para impressão e visualização através de um monitor de vídeo.

O grafo da esquerda é o responsável pela codificação da imagem, cuja captura é representada através do seu nó de entrada (*Cap*). Capturada, a imagem tem sua componente de luminância filtrada por um filtro passa-alta na escala de cinza (*Filtro*) e codificada em JPEG (*Comp. JPEG*). O armazenamento da imagem é representado pelo nó de saída do grafo (*Arm*).

O grafo da direita especifica a chegada ao sistema de uma imagem em JPEG (*Ent*), sua decodificação (*Dec. JPEG*) e as transformações do espaço de cores RGB para aqueles adequados à impressora (CYMK) e ao monitor de vídeo (YIQ). Os nós de entrada e saída dos grafos não são



a) Codificação b) Decodificação

Figura 7.15 Exemplo de Aplicação Real

tarefas do sistema e representam apenas interfaces com outros elementos, daí seus tempos de execução bem reduzidos. Assim, os prazos dos nós de saída representam os limites para que os dados sejam entregues aos sistemas de armazenamento, impressão e visualização acoplados. Todos os tempos de execução e comunicação na alocação de entrada são mostrados ao lado dos nós e arcos dos grafos da Figura 7.15, em milissegundos.

Como dito anteriormente, os desempenhos das tarefas do exemplo foram tomados de *benchmarks* do EEMBC, que os disponibiliza em iterações/segundo e considera imagens digitais com resolução de 320x240 *pixels* e 225 KBytes. Entre as informações fornecidas junto com eles, é dito que, devido à escalabilidade dos algoritmos empregados, os desempenhos das tarefas podem ser tomados inversamente proporcionais ao tamanho da imagem (em *pixels*) [35]. Desse modo, por considerarmos uma imagem com resolução de 640x480 *pixels* (900 KBytes), utilizamos no nosso exemplo os desempenhos dos *benchmarks* reduzidos em 4 vezes.

Pelo fato da taxa de compressão da tarefa *Comp. JPEG* depender da imagem de entrada, ela não se encontra entre os dados fornecidos pelo EEMBC junto ao *benchmark*. No entanto, por ser necessária para definir o tempo gasto nas comunicações entre as tarefas do sistema, empregamos uma taxa de compressão de 3 vezes no tamanho da imagem (em KBytes). Os tempos de comunicação foram tomados com base em taxas de transferência de *links* do pacote E3S.

Como já citado, as informações do EEMBC são relativas à processadores genéricos e à execução de tarefas em software. Uma vez que o objetivo do RECASTER é sintetizar redundância coletiva para tarefas em hardware e executando em PEs exclusivos, foi determinada uma alocação inicial com um processador para cada tarefa, e considerados os tempos de execução equivalentes a hardware dedicado. Essa consideração só foi possível pela utilização, tal como nos exemplos sintéticos, de fatores de desempenho para a obtenção dos tempos de execução das tarefas nos FPGAs. Os valores empregados foram os mesmos desses exemplos.

Embora esteja nos planos de expansão do EEMBC, não há ainda disponíveis dados relativos à implementações dos *benchmarks* em FPGAs, o que dificultou a composição de um exemplo mais preciso. Foram, no entanto, encontrados módulos IPs, especificamente para os dispositivos da Altera®, com algumas das aplicações desses *benchmarks*. O problema encontrado foi o fato desses IPs, além de serem projetados seguindo diretrizes distintas das empregadas nos *benchmarks*, apresentarem dados sobre sua implementação para poucos dispositivos da biblioteca de FPGAs utilizada nos exemplos desse capítulo. Mesmo entre esses, diferenças estruturais impedem a definição de um número único de ULs por tarefa em toda a biblioteca, como realizado nos exemplos sintéticos. Essa abordagem é válida quando a implementação das tarefas não utiliza recursos programáveis como blocos de memória embutidos, não disponíveis em todas as famílias de dispositivos da biblioteca. Entre as tarefas deste exemplo, apenas os IPs relativos às transformações de espaço de cores e filtragem são implementados somente com ULs e não apresentam esse problema. Os demais no entanto fazem uso de blocos de memória e de estruturas especiais para o processamento digital de sinais.

O problema da existência de múltiplos recursos de implementação nos FPGAs pode ser tratado pelo RECASTER mudando a variável taxa de ocupação das tarefas de um escalar para um vetor de dimensão igual ao maior número de possibilidades de recursos existente. O mesmo poderia ser realizado para a capacidade lógica dos dispositivos, substituindo-a, por exemplo, por um vetor [#_de_ULs #_de_bits_de_memória]. É importante lembrar porém que esse valor não influencia em nenhum procedimento realizado pelo algoritmo, uma vez que as taxas de ocupação das tarefas são suficientes para avaliar se um dispositivo pode ou não implementar um conjunto delas. Optamos, no entanto, por manter escalares as taxas de ocupação, obtendo-as a partir do número de ULs ocupados pelas tarefas em cada uma das famílias de FPGAs da biblioteca. Na

TAREFA	Número de ULs			
	FLEX10K	ACEX1K	APEX20K	APEXII
Filtro	**539**	539	539	539
Comp. JPEG	**8.260**	8.260	**7.660**	7.660
Dec. JPEG	**9.690**	9.690	**8.676**	8.676
RGB→YIQ (CYMK)	**358**	**358**	**380**	380

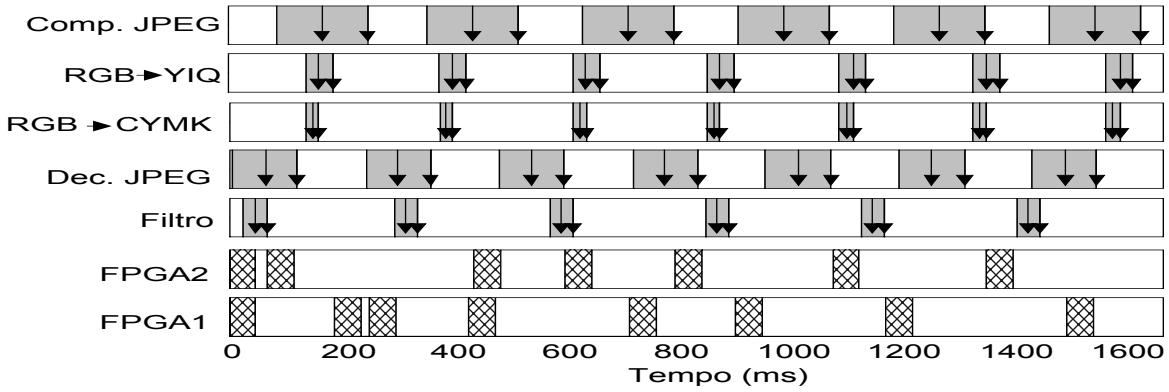
Tabela 7.14 Número de UL Ocupadas pelas Tarefas do Sistema

Tabela 7.14 são mostrados em negrito os valores obtidos na literatura. Os demais foram definidos levando em conta a semelhança existente entre os dispositivos das várias famílias.

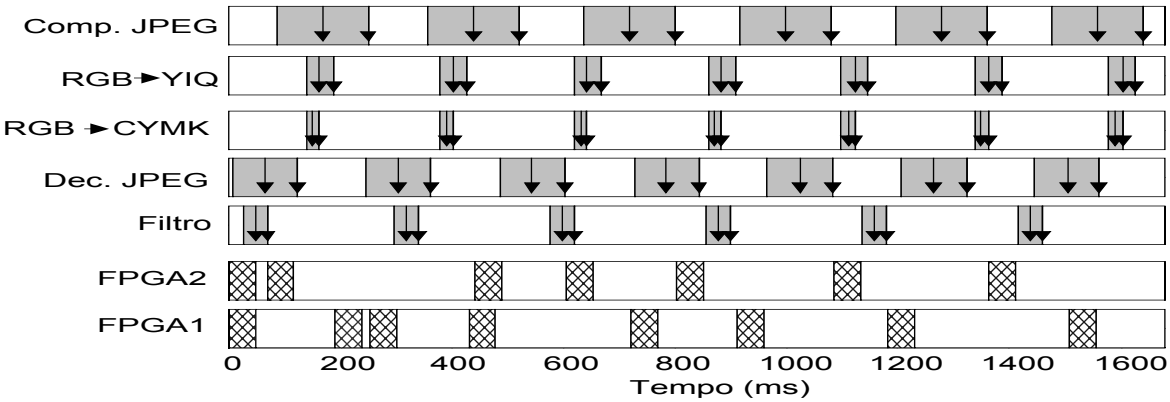
São apresentados na Tabela 7.15 os resultados obtidos na determinação da redundância coletiva do sistema nos dois critérios de otimização do algoritmo. Em ambos foram encontradas alocações compostas por dois FPGAs, de mesmos fatores de desempenho e capacidade lógica. Ambas diferem estruturalmente apenas no número de pinos programáveis e encapsulamento (área). Os dispositivos usados na alocação de menor área têm mais pinos que os dispositivos na alocação de menor custo, embora sejam menores que estes. Essa característica reflete-se na diferença entre seus custos: o custo da alocação de menor área é 31,25% maior que o da alocação de menor custo.

Critério de Otimização			Custo	Área
Tempo de Execução (s)			6,86	18,02
Solução Encontrada	Alocação	FPGA	EP20K300EQC2403	EP20K300EFC6723
		Fator de Desempenho	0.7497	0.7497
		Quantidade	2	2
		Custo (US$)	512,00	672,00
		Área (cm^2)	24,3	14,58
	Prazos	Violação Média (%)	14,75	14,75
		Violação Máxima (%)	47,76	47,76
	Média das Taxas de Ocupação dos FPGAs (%)		76,14	76,14

Tabela 7.15 Exemplo Real: Resultados Obtidos



a) Otimização de Custo



a) Otimização de Área

Figura 7.16 Escalonamentos dos Cenários ∅ das Soluções do Exemplo Real

Pelas Figuras 7.16a e 7.16b percebe-se que os escalonamentos dos cenários de ausência de falhas do sistema são os mesmos independente do critério de otimização adotado. Isso decorre da igualdade entre os tempos de reconfiguração e capacidades lógicas dos FPGAs das alocações, fatores que, junto com o número de pinos programáveis, determinam o escalonamento. Caso as tarefas tivessem quantidades significativas de pinos de interface, essas diferenças poderiam gerar escalonamentos diferentes para os diferentes critérios mostrados nas Figuras 7.16a e 7.16b.

Não foram realizadas comparações das soluções de redundância coletiva com as obtidas empregando-se duplicação simples de PEs pelo fato de terem sido usados na alocação inicial do sistema processadores de propósito geral, que geralmente apresentam encapsulamentos com área maior que circuitos dedicados. Comparações não seriam portanto justificáveis nesse caso.

Resumo do Capítulo 7

Neste capítulo foram apresentados os resultados computacionais empregados na avaliação do algoritmo RECASTER, obtidos através de um protótipo desenvolvido em linguagem C++. Na primeira seção foi caracterizada a biblioteca de FPGAs utilizada em todos os exemplos de avaliação, quatro sintéticos, gerados através da ferramenta TGFF, e um baseado numa aplicação de processamento digital de imagens.

Os três primeiros exemplos sintéticos foram empregados para demonstrar a capacidade de exploração de projeto do RECASTER sob variações de: a) requisito de recursos programáveis para as tarefas do sistema de entrada (em cada exemplo); b) complexidade do sistema (ao longo dos exemplos). Além disso, em cada um deles foram discutidos os resultados do algoritmo com a comparação das características fornecidas para as soluções de menor custo e área.

O quarto exemplo teve por objetivo apresentar aspectos relativos ao tempo de execução do algoritmo e seus principais fatores determinantes. Já o quinto e último exemplo, de aplicação real, serviu sobretudo para levantar questões sobre as dificuldades enfrentadas para se obter exemplos "reais" para *benchmark* dos algoritmos de síntese de sistemas embutidos.

A ausência de comparações dos resultados obtidos pelo RECASTER com os de outros algoritmos de síntese deve-se à inexistência, na literatura, de trabalhos com abordagem semelhante para reduzir a dimensão de sistemas embutidos tolerantes a falhas. Denominada tolerância a falhas através de redundância coletiva, essa abordagem foi proposta neste mesmo trabalho (capítulo 5), sendo portanto esperada a ausência de algoritmos semelhantes, pelo menos até o presente. De fato, o algoritmo foi desenvolvido para auxiliar no preenchimento do que acreditamos ser uma lacuna entre as formas de obtenção de sistemas tolerantes a falhas em hard-ware da atualidade. Os testes realizados neste capítulo foram portanto construídos e detalhados para evidenciar as potencialidades do algoritmo e a forma de explorá-lo eficientemente no projeto de um sistema embutido.

Capítulo 8

Conclusão

Neste capítulo são descritas as contribuições deste trabalho e enumeradas possíveis extensões.

8.1 Contribuições

Ferramentas de síntese no nível de sistema têm se mostrado cada vez mais necessárias para o projeto de sistemas embutidos, sobretudo os distribuídos. Nesse sentido, diversas metodologias têm sido propostas, a partir das quais são formulados algoritmos para sua implementação computacional. Essa diversidade decorre do aumento das aplicações dos sistemas embutidos, e das características de seus componentes. Esse trabalho explora o uso de hardware reconfigurável (componente) como elemento redundante compartilhado em sistemas embutidos tolerantes a falhas (aplicação). Tal alternativa para o projeto desses sistemas, o desenvolvimento de um algoritmo para auxiliar na sua síntese, e a implementação de uma ferramenta computacional, constituem as principais contribuições deste trabalho.

O algoritmo desenvolvido, denominado RECASTER, mostrou-se capaz de explorar a reconfigurabilidade de FPGAs para empregá-los como *redundância coletiva* de componentes de hardware dedicado de um sistema embutido. Através das margens de tempo do sistema, o algoritmo escalona as reconfigurações de FPGAs e execuções de tarefas concorrentemente, respeitando as restrições de tempo real especificadas. A determinação e uso dessas margens mostrou-se uma metodologia eficaz na realização desses escalonamentos, que tem como objetivo primário viabilizar uma alocação de FPGAs como redundância coletiva, o que é feito escalonando reconfigurações e impondo atrasos às tarefas apenas quando necessários.

8.2 Trabalhos Futuros

O trabalho descrito nessa dissertação pode ser ampliado em aspectos relacionados à:

- ARQUITETURA DE REDUNDÂNCIA COLETIVA
  - Aperfeiçoamento da arquitetura proposta, sobretudo da interface entre os FPGAs e os demais PEs do sistema;
  - Desenvolvimento de modelos para a determinação da confiabilidade e disponibilidade de sistemas que façam uso dessa arquitetura.

- MODELO DE FALHAS DO SISTEMA
  - Extensão do modelo de tratamento de falhas simples para falhas múltiplas, considerando cenários de falhas com mais de uma tarefa executando em FPGAs.

- ESCALONAMENTO DO SISTEMA
  - Determinação de margens de tempo a partir de escalonamentos *EARLY* e *LATE* obtidos com restrições de recursos aplicadas não só às tarefas, mas também aos eventos de comunicação.

- MODELO DE FPGA
  - Estudo da viabilidade de extensão do modelo de FPGA utilizado para o emprego de dispositivos com reconfiguração dinâmica parcial.

- IMPLEMENTAÇÃO COMPUTACIONAL
  - Aperfeiçoamento da interface de entrada de dados do algoritmo;
  - Aperfeiçoamento das estruturas de dados manipuladas pelo algoritmo, de forma à reduzir seu tempo de execução e a quantidade de memória necessária para executá-lo.

Referências Bibliográficas

[1] Alfke, P.,"The Future of Field-Programmable Gate Arrays," *Proc. of 5th Workshop on Electronics for LHC Experiments*, CERN Report CERN 99-09(1999).

[2] Altera, Corp., "Configuring SRAM-Based LUT Devices," Application Note 116, version 2.0, 2001.

[3] Altera, Corp., "Excalibur**:** Embedded Processor Programmable Solutions," June 2001.

[4] Altera, Corp., "Introduction to ISP," version 3.0, Feb. 1998.

[5] Altera, Corp., "Signal Processing IP Megafunctions: Signal Processing Solutions for System-on-a Programmable-Chip Designs", May 2001.

[6] Beasley, J. E., "Population Heuristics," Management School, Imperial College, London, England. 1999.

[7] Bertossi, A. A., Fusiello, A., Mancini, L. V.,"Fault-Tolerant Deadline-Monotonic Algorithm for Scheduling Hard-Real-Time Tasks," Proc. of 11th Intl. Parallel Processing Symp. , Geneva, Switzerland, Apr. 1997.

[8] Bianco, L., Auguin, M., Gogniat, G., Pegatoquet, A., "A Path Analysis based Partioning for Time Constrained Embedded Systems," In *6th Int. Workshop on Hardware/Software Co-Design*, 1998.

[9] Boulis, A., Srisvastava, M., "System Design of Active Basestations based on Dynamically Reconfigurable Hardware," *Proc. 37th Design Automation Conference*, Los Angeles, CA, June, 2000.

[10] Brown, S., Rose, J., "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers,* pp. 42-57, Summer Edition, 1996.

[11] Dave, B. P., Jha, N. K., "COFTA: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded System Architectures for Low Overhead Fault Tolerance," *Proc. Intl. Symp. Fault-Tolerant Computing*, pp. 339-348, June 1997.

[12] Dave, B. P., Jha, N. K., "COHRA: Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Embedded Systems," *IEEE Trans. on Computer-Aided Desig*n, vol. 17, pp. 900–919, Oct. 1998.

[13] Dave, B., "CRUSADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems," *Proc. of Design, Automation and Test in Europe Conf*., pp. 97–104, Mar. 1999.

[14] Dave, B., Lakshminarayana, G., Jha, N. K., "COSYN: Hardware-Software Co-Synthesis of Embedded Systems," *Proc. of Design Automation Conf*., pp. 703–708, June 1997.

[15] Dick, R. P., Jha, N. K., "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems," *Proc. of Int. Conf. on Computer-Aided Desig*n, pp. 62–68, Nov. 1998.

[16] Dick, R. P., Jha, N. K., "COWLS: Hardware-Software Co-Synthesis of Distributed Wireless Low-Power Embedded Client-Server Systems," *Proc. of Int. Conf. on VLSI Desig*n, pp. 114–120, Jan. 2000.

[17] Dick, R. P., Jha, N. K., "MOCSYN: Multiobjective Core-based Single-Chip System Synthesis," *Proc. of Design, Automation and Test in Europe Conf*., pp. 263–270, Mar. 1999.

[18] Dick, R. P., *Multiobjective Synthesis of Low-Power Real-Time Distributed Embedded Systems*. PhD thesis, Dept. of Electrical Eng., Princeton University, Nov. 2002.

[19] Dick, R. P., Rhodes, D. L., Wolf, W., "TGFF: Task graphs for free," *Proc. of Int. Workshop on Hardware/Software Co-Desig*n, pp. 97–101, Mar. 1998.

[20] Dick. R. P., Jha, N. K., "MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Distributed Embedded Systems," *IEEE Trans. on Computer-Aided Desig*n, vol. 17, pp. 920–935, Oct. 1998.

[21] Dömer, R., Gajski, D. D. "Reuse and Protection of Intellectual Property in the SpecC System," *Proc. of Asia & South Pacific Design Automation Conference*, Yokohama, Japan, Jan. 2000.

[22] Edwards, S., Lavagno, L., Lee, E. A., Sangiovanni-Vincentelli, A., "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proc. of IEE*E, pp. 366–390, Mar. 1997.

[23] Gajski, D. D., *Principles of Digital Design*. Prentice-Hall, Upper Saddle River, NJ, 1997.

[24] Gajski, D. D., Vahid, F., Narayan, S., Gong, J., *Specification and Design of Embedded System*s. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[25] Gajski, D., Vahid, F., Narayan, S., Gong, J., "SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, no. 1, pp. 84-100, Mar. 1998.

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

[27] *Gärtner, F. C.*, "Fundamentals of Fault Tolerant Distributed Computing in Asynchronous Environments," *ACM Computing Surveys*, 31 (1), pp. 1-26, Mar. 1999.

[28] Ghosh, S., Melhem, R., Mossé, D., Sarma, J. S.,"Fault-Tolerant Rate-Monotonic Scheduling," *Journal of Real-Time Systems*, vol 15, no. 2, Sept. 1998.

[29] Girault, A., Lavarenne, C., Sighireanu, M., Sorel, Y., "Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems," *Proc. of 21st Intl. Conf. on Distributed Computing* Systems, Phoenix, USA, apr. 2001.

[30] Hauser, J. R., *Augmenting a Microprocessor with Reconfigurable Hardware*. PhD thesis, Dept. of Computer Science, University of California, Berkeley, Fall 2000.

[31] Hou, J., Wolf, W., "Process Partitioning for Distributed Embedded Systems," *Proc. 4th Intl. Workshop on Hardware/Software Codesign, Codes/CASHE'9*6, pages 70 – 76, March 1996.

[32] http://helsinki.ee.princeton.edu/~dickrp/ - Informações do E3S. 12/2002.

[33] http://www.altera.com/ - HP para a obtenção de *datasheets* de FPGAs.

[34] http://www.arrow.com/ - HP para a consulta de preços de componentes. 02/2003.

[35] http://www.eembc.org/ - Informações do EEMBC. 12/2002.

[36] http://www.xilinx.com/xilinxonline/partreconfaq.htm#

[37] Jeong, B., Yoo, S., Lee, S., Choi, K.Y., "Hardware-Software Cosynthesis for Run-Time Incrementally Reconfigurable FPGAs," *Proc. Asia & South Pacific Design Automation Conference* ,pp.169-174, Jan 2000.

[38] Johnson, B. W, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, 1989.

[39] Kalavade and E. A. Lee, "A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem," *Proc. International Workshop on Hardware-software Co-desig*n, pp. 42-48, 1994.

[40] Kalavade, A., Lee, E. A., "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design and Test of Computer*s, vol. 3, pp. 16–28, Sept. 1993.

[41] Kwiat, K., Hariri, S., "Efficient Hardware Fault Tolerance Using Field-Programmable Gate Arrays," In *2nd International Society of Science and Applied Technologies (ISSAT) Int. Conf. on Reliability and Quality in Design*, 1995.

[42] Landis, D., *Programmable Logic and Application Specific Integrated Circuits*, Pennsylvania State University, 1997.

[43] Laprie, J. C., "Dependable Computing and Fault Tolerance: Concepts and Terminology". *Proc. of 15th IEEE Symposium on Fault Tolerant Computing Systems* , pp. 2-11, June 1985.

[44] Lawler, E. L., Martel, C. U., "Scheduling Periodically Occurring Tasks on Multiple Processors," *Information Processing Letter*s, vol. 7, pp. 9–12, Feb. 1981.

[45] Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., Stockwood, J., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures," *Proc. of 37th Design Automation Conference*, Los Angeles, CA, June, 2000.

[46] Maurer, S. S., "A Survey of Embedded Systems Programming Languages,". *IEEE Potencials*. April/May Edition. pp 30-34, 2002.

[47] Mei, B., Schaumont, P., Vernalde, S., "A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems," In *11th ProRISC workshop on Circuits, Systems and Signal Processing*, Veldhoven, Netherlands, Nov. 2000.

[48] Monteiro, L. H. J., *Elementos de Álgebra*. Ao Livro Técnico S. A., Rio de Janeiro, 1971.

[49] Narayan, S., Gajski, D. D., "Features Supporting System-Level Specification in HDLs,", pp. 540-545, In *European Design Automation Conferenc*e, 1993.

[50] Pimentel, A. D., Hertzberger L. O., Lieverse, P., Wolf, Deprettere, E., F., "Exploring Embedded-Systems Architectures with Artemis," *IEEE Computer*, pp. 58-63, 2001.

[51] Prakash, S., Parker, A., "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," *J. of Parallel & Distributed Computin*g, vol. 16, pp. 338–351, Dec. 1992.

[52] Rutenbar, R. A. (Panel Chair), "(When) Will FPGAs Kill ASICs?," *Proc. 38th Design Automation Conference*, Las Vegas, USA, June 2001.

[53] Shang, L., Jha, N. K., "Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs," *Proc. of Int. Conf. on VLSI Desig*n, pp. 345–352, Jan. 2002.

[54] Slomka, F., Dorfel, M., Munzenberger, R., Holfmann, R., "Hardware/Software Codesign and Rapid Prototyping of Embedded Systems," *IEEE Design & Test of Computers,* pp. 28-38, April-June 2000.

[55] Srinivasan, S., Jha, N. K., "Hardware-Software Co-Synthesis of Fault-Tolerant Real-Time Distributed Embedded Systems," *Proc. of European Design Automation Conf*., pp. 334–339, Sept. 1995.

[56] Vahid, F., Givargis, T., *Embedded System Design: A Unified Hardware/Software Approach*. Department of Computer Science and Engineering – University of California, 1999.

[57] Vital Information Publications, *Sensor Business Digest*. Vol. 11, No. 5, July 2002.

[58] Xilinx, Corp., "Architecting Systems for Upgradability with IRL (Internet Reconfigurable Logic)" *Application Note 412*, version 1.0, 2001.

[59] Xilinx, Corp., "Spartan-II FPGA Family Configuration and ReadBack," *Advanced Application Note 176*, version 0.9, 1999.

[60] Xilinx, Corp., "VIRTEX™ Configuration and ReadBack,". *Application Note 138*, version 1.0, 1999.

[61] Zipf, P., Bauer, C., Wojtkowiak, H, "A Hardware Extension to Improve Fault Handling In FPGA-based Systems," *Proc. of Design & Diagnostics of Electronic Circuits and Systems*, Slovakia, 2000.

Apêndice A

A.1 Grafos de Tarefas Gerados pelo TGFF

A.1.1 Primeiro Exemplo Sintético

A Figura A.1 mostra o grafo relativo à especificação funcional do primeiro exemplo sintético avaliado pelo RECASTER (Capítulo 7), tal como apresentado pelo TGFF. Esse tipo de saída, em formato .eps, é opcional e deve ser definida no seu arquivo de entrada.

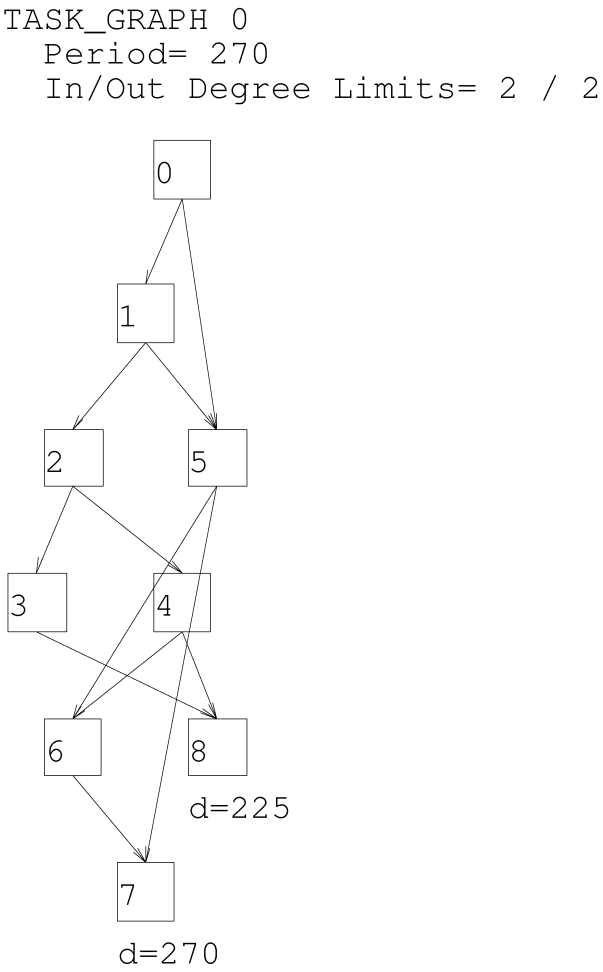


Figura A.1 Grafo do Primeiro Exemplo Sintético

A.1.2 Segundo Exemplo Sintético

Os dois grafos da Figura A.2 correspondem à especificação funcional do segundo exemplo sintético avaliado pelo RECASTER (Capítulo 7), tal como apresentada pelo TGFF. O limite de arcos de entrada e saída nos nós aparece abaixo dos períodos dos grafos. Por ter sido requisitada a geração de um sistema multi-taxa para o exemplo, um dos grafos é formado com o dobro do número de tarefas do outro. Isso ocorre porque a ferramenta considera um valor médio de tempo de execução de tarefas, mais tempos de comunicação, para gerar os prazos associados aos nós de saída (d's).

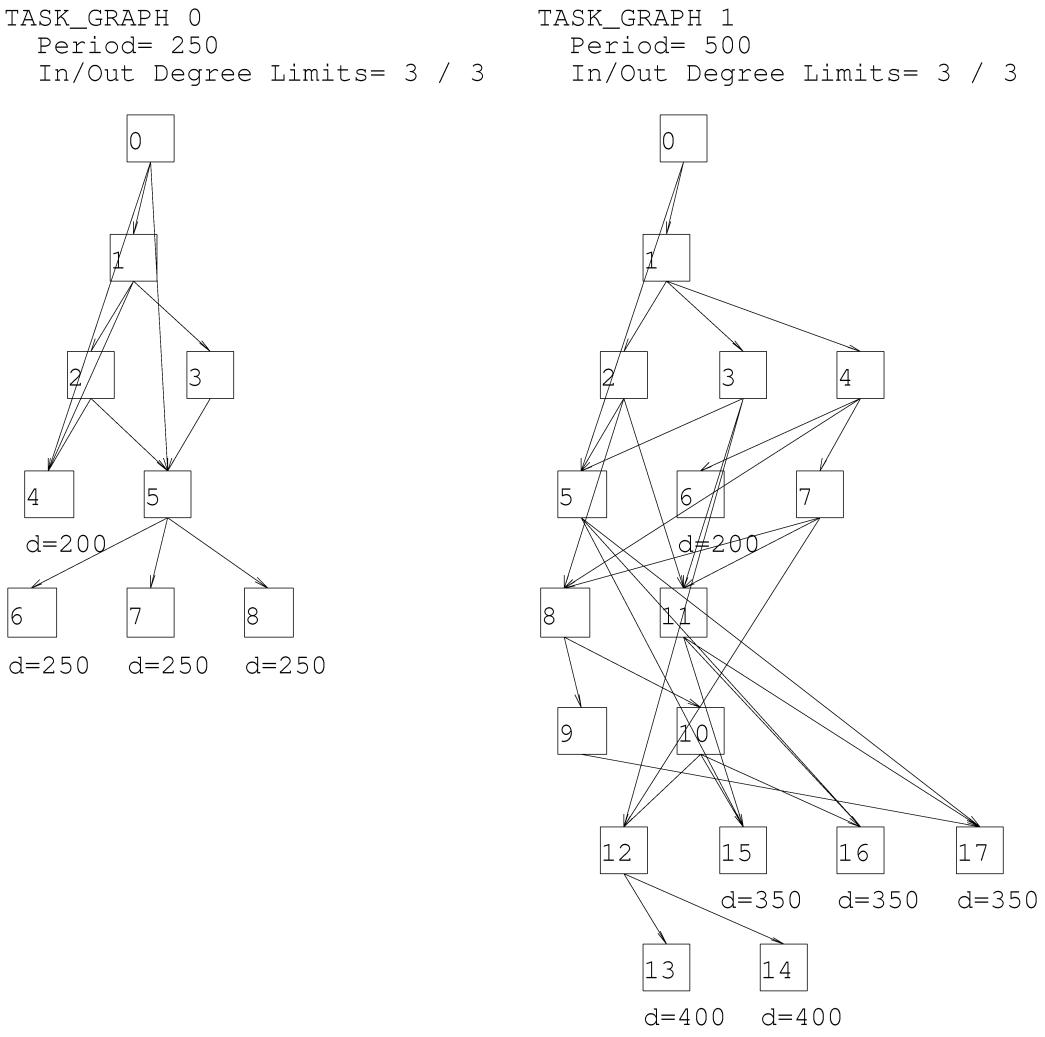


Figura A.2 Grafos do Segundo Exemplo Sintético

A.1.3 Terceiro Exemplo Sintético

Figura A.3 Grafos do Terceiro Exemplo Sintético

A.1.4 Quarto Exemplo Sintético

Os grafos dos sistemas de entrada utilizados na avaliação dos tempos de execução do algoritmo têm variações irregulares de profundidade, apesar do aumento regular de seus números de tarefas. Essas irregularidades refletem no número de tarefas por "linha" dos grafos, ou seja, sua "largura". Devido as dificuldades encontradas na apresentação gráfica dessas características, apresentando os grafos tal como fornecidos pelo TGFF, optamos por listar suas profundidades e larguras média e máxima como forma de evidenciar tais diferenças. A Figura A.4 mostra, para um grafo genérico, o que são profundidade, linhas e larguras de linha.

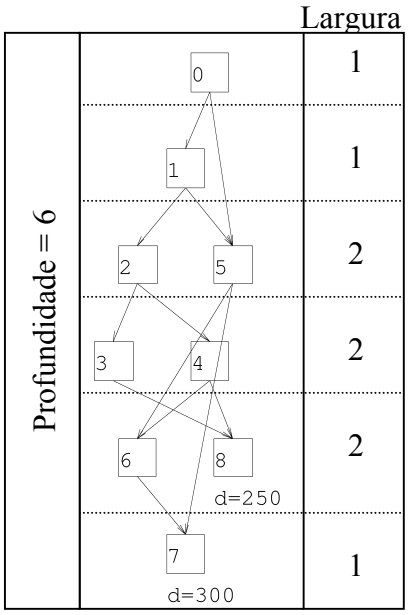


Figura A.4 Caracterização de um Grafo

# de Tarefas		10	20	30	40	50	60	70	80	90	100	125	150	175	200
Profundidade		6	12	15	16	18	21	22	24	26	27	29	32	33	34
Largura	Média	1,66	1,66	2,00	2,50	2,77	2,85	3,18	3,33	3,46	3,70	4,31	4,68	5,30	5,88
	Máxima	2	2	4	6	6	7	8	8	8	9	9	10	10	11

Tabela A.1 Características dos Grafos do Exemplo 4