Escalonamento de Tarefas Imprecisas em Ambiente Distribuído

Rômulo Silva de Oliveira

Laboratório de Controle e Microinformática - LCMI
Universidade Federal de Santa Catarina
Caixa Postal 476 - 88040-900 - Florianópolis-SC
Brasil
Telefone: +55 (48) 231-9202 - Fax: +55 (48) 234-9770
e-mail: romulo@lcmi.ufsc.br

20 de Janeiro de 1997

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ESCALONAMENTO DE TAREFAS IMPRECISAS EM AMBIENTE DISTRIBUÍDO

TESE SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA PARA OBTENÇÃO DO GRAU DE DOUTOR EM ENGENHARIA, ÁREA ENGENHARIA ELÉTRICA, ÁREA DE CONCENTRAÇÃO SISTEMAS DE INFORMAÇÃO.

RÔMULO SILVA DE OLIVEIRA

FLORIANÓPOLIS, 20 DE JANEIRO DE 1997.

Resumo

Sistemas computacionais de tempo real são identificados como aqueles sistemas submetidos a requisitos de natureza temporal. Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto.

Um problema básico encontrado na construção de sistemas distribuídos de tempo real é a alocação e o escalonamento das tarefas nos recursos computacionais disponíveis. Existe uma dificuldade intrínsica em compatibilizar dois objetivos fundamentais: garantir que os resultados serão produzidos no momento desejado e dotar o sistema de flexibilidade para adaptar-se a um ambiente dinâmico e, assim, aumentar sua utilidade.

Uma das técnicas existentes na literatura para resolver o problema de escalonamento tempo real é a Computação Imprecisa. Nesta técnica, cada tarefa da aplicação possui uma parte obrigatória e uma parte opcional. A parte obrigatória é capaz de gerar um resultado com a qualidade mínima, necessária para manter o sistema operando de maneira segura. A parte opcional refina este resultado, até que ele alcançe a qualidade desejada. Esta técnica procura conciliar os dois objetivos fundamentais citados antes. Entretanto, não existe na literatura um estudo amplo sobre a questão de "como resolver o problema do escalonamento quando sistemas de tempo real distribuídos são construidos a partir do conceito de Computação Imprecisa".

O objetivo geral desta tese é mostrar como aplicações de tempo real, construídas a partir do conceito de Computação Imprecisa, podem ser escalonadas em ambiente distribuído. Em outras palavras, mostrar que o conceito de Computação Imprecisa pode ser adaptado para um ambiente onde tarefas executam em diferentes processadores e a comunicação entre elas é implementada através de mensagens. É mostrado que o problema proposto pode ser dividido em quatro problemas específicos. São eles:

- Como garantir que as partes obrigatórias das tarefas serão concluídas antes dos respectivos deadlines, em um ambiente onde tarefas podem executar em diferentes processadores e o emprego de mensagens cria relações de precedência entre elas.

- Como determinar que a execução de uma parte opcional não irá comprometer a execução das partes obrigatórias, previamente garantidas.

- Como escolher quais partes opcionais devem ser executadas, na medida em que o recurso "tempo de processador disponível" não permite a execução de todas elas.

- Como resolver qual tarefa executa em qual processador, de forma que todas as partes obrigatórias das tarefas possam ser garantidas e que as partes opcionais estejam distribuídas de forma que sua chance de execução seja maximizada.

Nesta tese são apresentadas soluções de escalonamento para estes quatro problemas específicos. Desta forma, o texto mostra que efetivamente Computação Imprecisa pode ser usada como base para a construção de aplicações distribuídas de tempo real.

Abstract

Real-time computing systems are defined as those systems subjected to timing constraints. In those systems, results must be not only logically correct but they also must be generated at the right moment.

A basic problem one finds when building a distributed real time system is the allocation and scheduling of tasks on the available computing resources. There is an intrinsic difficulty in simultaneously achieving two fundamental goals: to guarantee that results are generated by the desired time and to make the system flexible enough so it can adapt to a dynamic environment and, that way, increase its own utility.

The Imprecise Computation technique has been proposed in the literature as an approach to the scheduling of real-time systems. When this technique is used, each task has a mandatory part and an optional part. The mandatory part is able to generate a minimal quality result that is barely good enough to keep the system in a safe operational mode. The optional part refines the result until it achieves the desired quality level. This technique tries to conciliate the two fundamental goals mentioned above. Meanwhile, there is not in the literature a broad study on "how to solve the scheduling problem when real-time distributed systems are built based on Imprecise Computation concepts."

The overall goal of this theses is to show how real-time applications, that are built upon Imprecise Computation concepts, can be scheduled in a distributed environment. We intend to show that Imprecise Computation concepts can be adapted to an environment where tasks execute in different processors and communication among them is done by sending messages. It is shown in the text that we can split this problem in the following four specific problems:

- How to guarantee that mandatory parts will be finished before or at the respective task deadline, when we consider that tasks can execute in different processors and the use of messages creates precedence relations among them.

- How to know that the execution of an optional part will not jeopardize the execution of previously guaranteed mandatory parts.

- How to chose which optional parts should be executed when the resource "available processor time" is not enough to execute all of them.

- How to decide which task runs on which processor, in a way that all mandatory parts can be guaranteed and that optional parts are evenly spread over the system so as to maximize the chance they get actually executed.

This theses presents scheduling solutions for those four specific problems. In this way, the text shows that Imprecise Computation can effectively be used as the conceptual base for the construction of distributed real-time applications.

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

1.1 Motivação

Sistemas computacionais de tempo real (STR) são identificados como aqueles sistemas computacionais submetidos a requisitos de natureza temporal. Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. As falhas de natureza temporal nestes sistemas são, em alguns casos, consideradas críticas no que diz respeito às suas consequências.

Na medida em que o uso de sistemas computacionais prolifera em nossa sociedade, aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Estas aplicações variam muito com relação ao tamanho, complexidade e criticalidade. Entre os sistemas mais simples estão os controladores embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade deste espectro estão os sistemas militares de defesa e o controle de tráfego aéreo. Exemplos de aplicações críticas são os sistemas responsáveis pelo monitoramento de pacientes em hospitais e os sistemas embarcados em veículos, de automóveis até aviões e sondas espaciais. Entre aplicações não críticas estão os videogames e as aplicações multimedia em geral.

No contexto da automação industrial, são muitas as possibilidades (ou necessidades) de empregar sistemas com requisitos de tempo real ([REM 93]). Exemplos são os sistemas de controle embutidos em equipamentos industriais, os sistemas de supervisão e controle de células de manufatura e os sistemas responsáveis pela supervisão e controle de plantas industriais como um todo.

Em função do baixo custo dos processadores, dos avanços na área de redes de computadores e da necessidade física de algumas aplicações, soluções distribuídas são cada vez mais empregados. Nestes sistemas as tarefas da aplicação encontram-se distribuídas por diversos processadores, os quais são interconectados por uma rede de comunicação.

Um problema básico encontrado na construção de sistemas distribuídos de tempo real é a alocação e o escalonamento das tarefas nos recursos computacionais disponíveis. Existe uma dificuldade intrínsica em compatibilizar dois objetivos fundamentais ([BUR 91]): garantir que os resultados serão produzidos no momento desejado e dotar o sistema de flexibilidade para adaptar-se a um ambiente dinâmico e, assim, aumentar sua utilidade.

Em um extremo existem soluções de escalonamento que supõe um conjunto fixo de tarefas a serem executadas. Estas soluções reservam recursos para o pior caso e são capazes de garantir que todas as tarefas serão concluídas no momento correto. Entretanto, aplicações construidas desta forma resultam em sistemas pouco flexíveis e na subutilização dos recursos computacionais. No outro extremo temos as soluções de escalonamento que não garantem o comportamento temporal da aplicação. Tarefas são escalonadas na medida do possível. Embora os recursos computacionais sejam plenamente utilizados e o sistema resultante seja bastante flexível, a falta de uma garantia prévia para o seu comportamento temporal inviabiliza este tipo de solução para muitas classes de aplicações.

A dificuldade de escalonar tarefas com requisitos de tempo real é bastante conhecida, constituindo uma área de pesquisa intensa atualmente. Uma das técnicas

existentes na literatura para resolver o problema de escalonamento tempo real é a Computação Imprecisa. Esta técnica procura, de certa forma, conciliar os dois objetivos fundamentais citados antes.

Na Computação Imprecisa ([LIU 94]) cada tarefa da aplicação possui uma parte obrigatória ("mandatory") e uma parte opcional ("optional"). A parte obrigatória da tarefa é capaz de gerar um resultado com a qualidade mínima, necessária para manter o sistema operando de maneira segura. A parte opcional refina este resultado, até que ele alcançe a qualidade desejada. O resultado da parte obrigatória é dito impreciso ("imprecise result"), enquanto o resultado das partes obrigatória+opcional é dito preciso ("precise result"). Uma tarefa é chamada de tarefa imprecisa ("imprecise task") se for possível decompo-la em parte obrigatória e parte opcional.

Poucos trabalhos existentes na literatura tratam de Computação Imprecisa em ambiente distribuído. Quando o fazem, abordam questões localizadas e casos particulares. Não existe na literatura um estudo amplo sobre a questão de "como resolver o problema do escalonamento quando sistemas de tempo real distribuídos são construidos a partir do conceito de Computação Imprecisa".

1.2 Objetivos da Tese

O objetivo geral desta tese é mostrar como aplicações de tempo real, construídas a partir do conceito de Computação Imprecisa, podem ser escalonadas em ambiente distribuído. Em outras palavras, mostrar que o conceito de Computação Imprecisa pode ser adaptado para um ambiente onde tarefas executam em diferentes processadores e a comunicação entre elas é implementada através de mensagens.

No capítulo 3 será mostrado que, para alcançar o objetivo da tese, é necessário resolver especificamente quatro problemas:

- Como garantir que as partes obrigatórias das tarefas serão concluídas antes dos respectivos deadlines, em um ambiente onde tarefas podem executar em diferentes processadores e o emprego de mensagens cria relações de precedência entre elas.

- Como determinar que a execução de uma parte opcional não irá comprometer a execução das partes obrigatórias, previamente garantidas, violando as premissas usadas para fornecer tal garantia.

- Como escolher quais partes opcionais devem ser executadas, na medida em que o recurso "tempo de processador disponível" não permite a execução de todas as partes opcionais da aplicação.

- Como resolver qual tarefa executa em qual processador, de forma que todas as partes obrigatórias das tarefas possam ser garantidas e que as partes opcionais estejam distribuídas de forma que sua chance de execução seja maximizada.

Ao resolver estas quatro questões, estaremos mostrando que efetivamente Computação Imprecisa pode ser usada como base para a construção de aplicações distribuídas de tempo real.

1.3 Adequação às Linhas de Pesquisa do Curso

Entre os interesses do Laboratório de Controle e Microinformática (LCMI), pertencente ao Departamento de Engenharia Elétrica da Universidade Federal de Santa Catarina (EEL-UFSC), então os sistemas de tempo real. Especificamente, metodologias de desenvolvimento, ferramentas, linguagens de programação, algoritmos, sistemas operacionais e técnicas de descrição formal. No LCMI, estes assuntos já foram tema de projetos, trabalhos de graduação, dissertações de mestrado e teses de doutorado.

O trabalho descrito nesta proposta de tese está dentro do tema geral "escalonamento de sistemas computacionais com requisitos de tempo real". Logo, ele está perfeitamente integrado com as atividades do laboratório e do Curso de Pós-Graduação em Engenharia Elétrica desta universidade.

1.4 Organização do Texto

Esta tese está dividida em 8 capítulos. O capitulo 1 descreveu o contexto geral no qual o trabalho está inserido. Também foram apresentados os objetivos da tese e foi afirmada sua adequação às linhas de pesquisa do curso.

O capítulo 2 trata de sistemas tempo real em geral e o seu escalonamento. São definidos os conceitos básicos de tempo real e apresentadas as variações existentes nos modelos de tarefas adotados. As principais abordagens para o problema de escalonamento tempo real são descritas e classificadas. São discutidas as vantagens e desvantagens de cada abordagem. Também neste capítulo a técnica de Computação Imprecisa é descrita. São apresentadas motivações para seu emprego, formas de programação e as variações encontradas na literatura com respeito ao objetivo dos algoritmos de escalonamento baseados nesta técnica.

O capítulo 3 descreve o modelo de tarefas adotado, o problema a ser atacado e a abordagem que será utilizada para resolve-lo. Em particular, este capítulo mostra que o problema pode ser dividido em quatro componentes. Cada um destes quatro componentes será tratado especificamente em um dos capítulos seguintes da tese.

O capítulo 4 discute o problema da garantia para as partes obrigatórias em um ambiente distribuído. É desenvolvido um teste de escalonabilidade apropriado para situações onde existem relações de precedência entre tarefas, situação típica em sistemas distribuídos.

O capítulo 5 mostra como o tempo de processador não usado pelas partes obrigatórias pode ser usado para executar partes opcionais. Um teste de aceitação é empregado para determinar quando uma parte opcional pode ser executada. Um algoritmo existente na literatura é adaptado para o contexto de interesse, ou seja, tarefas imprecisas em ambiente distribuído.

O capítulo 6 trata da política de admissão para partes opcionais. São descritas heurísticas capazes de selecionar quais partes opcionais devem ser escalonadas em face aos limitados recursos disponíveis. São admitidas partes opcionais de forma a maximizar a utilidade do sistema como um todo.

O capítulo 7 aborda a questão da alocação de tarefas imprecisas em ambiente distribuído. O método de pesquisa aleatória conhecido como recozimento simulado ("simulated annealing") é empregado para efetuar a alocação de tarefas de tal forma que as partes obrigatórias possam ser garantidas e as partes opcionais possuam uma chance razoável de serem executadas.

Finalmente, o capítulo 8 contém as considerações finais sobre o trabalho desenvolvido. São destacados os principais resultados alcançados e identificadas questões ligadas à Computação Imprecisa que permanecem ainda em aberto na literatura. Estas questões poderão servir de ponto de partida para futuros trabalhos.

2 Sistemas Tempo Real e Escalonamento

2.1 Introdução

Sistemas computacionais de tempo real (STR) são identificados como aqueles submetidos a requisitos de natureza temporal. Em geral, requisitos temporais são expressos através de deadlines (prazo máximo para execução) associados com as reações do sistema a estímulos externos. A dificuldade de escalonar tarefas com requisitos de tempo real é bastante conhecida, constituindo uma área de pesquisa intensa. As falhas de natureza temporal são, em alguns sistemas, consideradas críticas no que diz respeito às suas consequências.

Nos sistemas em geral (que não são do tipo tempo real), a única preocupação é com a qualidade dos resultados. Embora uma execução rápida seja desejável, a abordagem é sempre do tipo "fazer o trabalho usando o tempo que for necessário". Sistemas tempo real possuem uma abordagem diferente, pois o tempo é limitado. É preciso garantir que será possível atender aos prazos, geralmente impostos pelo ambiente do sistema. Logo, a preocupação é "fazer o trabalho usando o tempo disponível".

A problemática dos sistemas tipo tempo real pode ser analisada a partir de diferentes perspectivas ou pontos de vista. Em cada ponto de vista, alguns aspectos são considerados em detalhe, enquanto outros são ignorados. Nesta tese a perspectiva predominante será aquela relacionada com o escalonamento. Este ponto de vista trabalha basicamente com recursos (processador, seção crítica, meio de comunicação, etc) e usuários de recursos (processos, métodos de objetos, mensagens, etc). O objetivo é definir as propriedades dos recursos, as propriedades dos usuários de recursos e os algoritmos de alocação e escalonamento utilizados para resolver os conflitos e atender os requisitos da aplicação. Em especial, os requisitos de caráter temporal. Nesta perspectiva, não importa se o usuário do recurso teve origem em um método de objeto ou uma cláusula prolog. Inclusive, a mesma solução de escalonamento pode ser usada para sistemas criados a partir de paradigmas de programação diferentes, desde que a solução considerada consiga escalonar corretamente os recursos computacionais durante a execução da aplicação.

No seu clássico trabalho [STA 88], Stankovic analisa as diversas concepções erradas associadas com sistemas de tempo real e discute os desafios a serem vencidos nesta área de pesquisa. Uma discussão do conceito "tempo", particularmente quando aplicado a sistemas em software, pode ser encontrada em [MOT 93]. Referências adicionais a trabalhos publicados na área de tempo real podem ser encontradas em [AUD 90c], [RAM 94] e [SHI 94].

2.2 Conceitos Básicos

Esta seção define alguns conceitos básicos ligados a sistemas de tempo real. Embora não exista um consenso universal no que diz respeito à terminologia de tempo real, diversos termos serão definidos nesta seção segundo o seu emprego mais comum. O restante da tese emprega à terminologia como definida nesta seção. Textos semelhantes podem ser encontrados em [AUD 90b], [AUD 90c], [LAN 90], [KOP 92], [MAG 92], [RAM 94] e [SHI 94].

2.2.1 Modelo de Tarefas e Características Temporais

Na maioria das vezes, sistemas de tempo real são descritos através de um modelo de execução baseado em tarefas ("tasks"). Normalmente, o termo tarefa está associado à unidade de concorrência em um sistema. Tarefas recebem opcionalmente dados, executam um algoritmo específico e geram algum tipo de saída. A tarefa estará logicamente correta se gerar sempre uma saída correta em função dos dados de entrada. Este é o conceito clássico da computação, válido para qualquer tipo de sistema em software. Quando a tarefa gera uma saída errada, ocorre uma falta lógica da tarefa. Em sistemas tempo real, além da correção lógica existe a necessidade de correção temporal. Uma tarefa estará temporalmente correta se gerar a saída dentro de um prazo satisfatório. Uma tarefa estará correta se estiver logicamente e temporalmente correta, ou seja, se sempre gerar uma saída correta dentro de um prazo satisfatório.

A definição de "saída correta" e "prazo satisfatório" dependem de cada aplicação em particular. A forma mais usual para especificar o prazo satisfatório é através de um deadline. O deadline corresponde ao momento máximo para a conclusão da tarefa. A princípio, toda tarefa deve ser concluída antes de seu deadline.

Existem dois tipos de situações onde uma tarefa é habilitada para executar. A situação mais usual ocorre quando uma tarefa termina e, como parte de sua saída, ativa outra tarefa ("event-trigged"). Uma tarefa também pode ser habilitada pela passagem do tempo ("time-trigged"). Com respeito à periodicidade da ativação, as tarefas podem ser classificadas como periódicas ("periodic") ou aperiódicas ("aperiodic"). Uma tarefa **T** é dita periódica se a cada período **P** de tempo ocorrer sempre uma ativação desta tarefa **T**. Caso contrário, a tarefa é chamada de aperiódica. As tarefas esporádicas ("sporadic") formam um subconjunto das tarefas aperiódicas. Uma tarefa aperiódica é também esporádica se existir um intervalo mínimo de tempo **I** (maior que zero) entre duas ativações sucessivas.

O momento no qual o escalonador toma conhecimento da necessidade de executar uma tarefa aperiódica (esporádica ou não) é chamado de momento de chegada ("arrival time"). No caso das tarefas periódicas, o início de cada período é considerado um momento de chegada.

O momento de liberação ("release time") para uma ativação individual corresponde ao instante a partir do qual a tarefa pode ser executada. Tarefas aperiódicas podem estar prontas para executar a partir da sua chegada (momento de liberação igual ao momento da chegada) ou não. Neste último caso, a partir do momento da chegada o escalonador tem conhecimento de que deverá executar a tarefa, mas somente a partir da sua liberação.

O tempo de resposta ("response time") associado com uma ativação em particular de uma tarefa é definido como a duração do intervalo de tempo compreendido entre a chegada da tarefa e o término de sua execução. O tempo máximo de resposta de uma tarefa ("maximum response time") é o maior tempo de resposta que esta tarefa poderá apresentar dentro da aplicação em questão. Logo, o tempo máximo de resposta é um limite máximo ("upper bound") para os tempos de resposta exibidos pela tarefa dentro da aplicação. O tempo máximo de resposta de uma tarefa não é uma propriedade definida apenas pela tarefa em questão, mas sim uma propriedade desta tarefa dentro de uma aplicação específica. As demais tarefas da aplicação podem interferir com a execução da tarefa em questão e, desta forma, aumentar o seu tempo máximo de resposta.

Tipicamente, cada tarefa é descrita temporalmente através dos seguintes valores:

- Período **P** (tarefas periódicas);
- Intervalo mínimo de tempo entre ativações **I** (tarefas esporádicas);
- Tempo máximo de execução **C**, considerando um processador específico;
- Deadline **D**, prazo máximo para concluir a execução da tarefa.

A forma exata de **P**, **I**, **C** e **D** depende da abordagem empregada na construção do sistema. Por exemplo, o tempo de execução **C** pode ser um valor constante, representando o pior caso. Ou ainda, **C** pode representar um par (C_1,C_2), onde C_1 é o tempo de computação necessário dentro do prazo de execução e C_2 é o tempo das computações da tarefa que podem ser realizadas fora (depois) do prazo de execução.

Algumas aplicações definem ainda um *jitter*[1] de terminação ("maximum jitter") para cada tarefa periódica. Ainda que uma tarefa periódica seja concluída sempre antes do deadline, podem existir variações no intervalo de tempo entre duas conclusões consecutivas. Isto decorre do fato da tarefa poder ser concluída em um instante mais ou menos próximo do deadline. Se o instante exato de conclusão da tarefa puder ser observado, de alguma forma, em um osciloscópio, aparecerá na tela um certo tremor ou variação. Este tremor ou variação é chamado de *jitter* de terminação. Alguns algoritmos para controle de processos são sensíveis ao nível de *jitter* de terminação que a tarefa experimenta.

Um conceito semelhante ao *jitter* de terminação é o *jitter* de liberação ("release jitter"). Isto acontece sempre que existem tarefas esporádicas mas o sistema somente amostra eventos externos em momentos pré-determinados. Suponha uma tarefa esporádica **T** com intervalo mínimo **I** entre ativações. Suponha ainda que o sistema somente amostre os eventos externos a cada **J** unidades de tempo. Se o evento associado com a liberação da tarefa **T** ocorrer imediatamente após uma amostragem, ele somente será reconhecido, para efeitos de escalonamento, na próxima amostragem. Este atraso torna possível que dois eventos consecutivos sejam reconhecidos pelo sistema no intervalo de tempo **I-J**, menor que o valor suposto **I**. Nesta situação, é dito que esta tarefa possui um *jitter* de liberação de **J** unidades de tempo. Também ocorre *jitter* na liberação de tarefas que dependem da recepção de mensagens para iniciar sua execução.

O conceito de instante crítico ("critical instant") para um conjunto de tarefas foi definido em [LIU 73] como sendo o instante no tempo quando todas as tarefas do conjunto estão prontas para executar simultaneamente. Os testes de escalonabilidade em alguns algoritmos de escalonamento se utilizam do conceito de instante crítico para determinar se um conjunto de tarefas é ou não escalonável, em uma análise de pior caso. O instante crítico representa o momento no qual existe a maior demanda por processador, durante a execução de um sistema.

Algumas propostas encontradas na bibliografia agrupam as tarefas em atividades ("activities" ou "transactions"). Uma atividade é composta por um conjunto de tarefas que mantêm relações de precedência entre si. Se a tarefa $\mathbf{T_i}$ precede a tarefa $\mathbf{T_j}$, então a tarefa

---

[1]A palavra inglesa "jitter", como empregada na literatura de tempo real, poderia ser traduzida como "tremor" ou "variação". Entretanto, mesmo na literatura brasileira, o termo jitter é normalmente utilizado sem ser traduzido. Por uma questão de clareza, neste texto será mantida a palavra em inglês jitter.

$\mathbf{T_j}$ somente pode iniciar sua execução após o término da tarefa $\mathbf{T_i}$. Em geral, o início da atividade está associado a um estímulo recebido pelo sistema. O estímulo pode ser a ocorrência de um evento no ambiente ou a simples passagem do tempo. Da mesma forma, o final da atividade está geralmente associado a uma resposta do sistema para o ambiente.

Uma questão chave é como expressar o deadline para execução de uma tarefa. A situação mais simples é ilustrada na figura 2.1. Em sistemas pequenos (por exemplo, um controlador lógico programável), existe uma ligação direta entre ambiente e tarefa. Quando o ambiente gera um determinado estímulo (evento ou passagem de tempo), o sistema ativa uma tarefa. Esta tarefa executa e sua saída é a resposta do sistema ao ambiente. Neste caso, o deadline para a execução da tarefa é definido pelo ambiente do sistema. Ele deverá estar descrito na especificação do sistema ou ter sido derivado a partir da especificação.

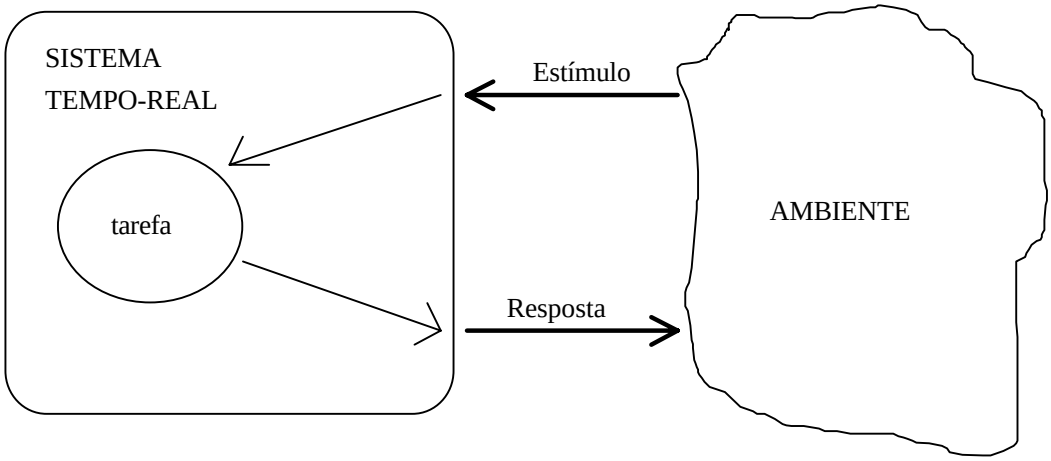

Figura 2.1 - Sistema simples, uma única tarefa responde ao ambiente.

Em sistemas maiores, especialmente os distribuídos, definir os deadlines das tarefas é mais complicado. A figura 2.2 ilustra uma situação que pode ocorrer, por exemplo, em uma planta industrial envolvendo controle hierarquizado. Neste caso, um estímulo do ambiente ativa uma cadeia de tarefas dentro do sistema, em diferentes computadores. O trabalho cooperativo destas tarefas, possivelmente em processadores distintos, vai produzir no final a resposta do sistema. A especificação de um sistema é feita em termos do comportamento observável deste sistema, ou seja, a especificação do sistema descreve as restrições temporais em termos de estímulo e resposta. Internamente, o projetista é livre para construir a solução que julgar melhor. Pode existir um conjunto enorme de projetos diferentes que atendem os requisitos da especificação e, portanto, resultam em sistemas corretos. Isto dá margem a diferentes maneiras de expressar o "prazo satisfatório" de uma tarefa.

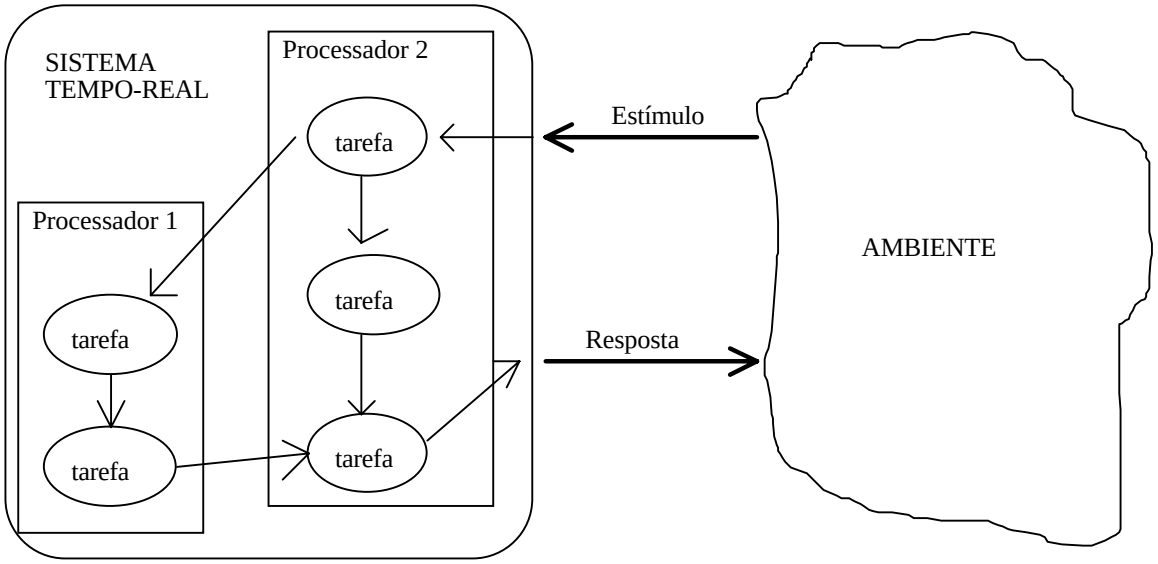

Figura 2.2 - Sistema complexo, um grafo de tarefas responde ao ambiente.

2.2.2 Conceitos Ligados ao Escalonamento

Esta seção discute alguns conceitos básicos de sistemas tempo real que estão ligados ao escalonamento.

Escalonamento, Escalonador e Escala de execução

O termo escalonamento ("scheduling") identifica o processo, como um todo, de alocar recursos às tarefas. Uma solução de escalonamento mostra como o problema de alocar recursos às tarefas pode ser abordado ou mesmo resolvido. O escalonamento pode ser feito tanto em tempo de projeto como em tempo de execução. Também existem soluções mistas, onde o escalonamento é feito parte em tempo de projeto e parte em tempo de execução.

O escalonador ("scheduler") é o componente do sistema responsável, em tempo de execução, pela gerência dos recursos considerados. É o escalonador quem implementa, durante a execução, a solução de escalonamento do sistema. O papel do escalonador no sistema varia muito, conforme a solução de escalonamento adotada.

Uma escala de execução ("schedule") é uma lista ou tabela que descreve quando cada uma das tarefas de um determinado conjunto vai ocupar determinado recurso. Na maioria das vezes, o recurso em questão é um processador. Algumas escalas de execução simplesmente ordenam as tarefas (escalonamento não preemptivo). Neste caso, cada tarefa deverá ocupar o recurso na ordem indicada pela escala, tão logo a tarefa anterior o libere. Existem escalas de execução mais complexas, onde a execução da tarefa não acontece de forma continua, mas sim particionada (escalonamento preemptivo). Neste caso, a escala descreve execuções parcias das tarefas, espalhadas ao longo do tempo.

Uma grade ("time grid") é um tipo especial de escala de execução. A grade contém uma sequência finita de slots de tempo ("time slots"). Todos os slots de tempo possuem a mesma duração. Ela indica qual tarefa executa em quais slots de tempo. Durante a

execução, o escalonador aplica a grade ciclicamente, selecionando para execução em cada slot de tempo a tarefa indicada na grade.

Previsibilidade determinista e Previsibilidade probabilista

O termo previsibilidade ("predictability") é utilizado para descrever a capacidade de se conhecer o comportamento temporal de um sistema antes de sua execução, em função do escalonamento empregado. Especificamente, saber em tempo de projeto se as tarefas serão executadas dentro dos deadlines. Na literatura, a noção de previsibilidade é associada com uma antecipação determinista (todos os deadlines serão cumpridos) ou com uma antecipação probabilista (qual a probabilidade de um deadline ser cumprido) para o comportamento temporal.

Teste de escalonabilidade e Cálculo da escala de execução

O processo de escalonamento de um conjunto de tarefas é, muitas vezes, dividido em duas etapas. Inicialmente, um teste de escalonabilidade determina se é possível atender aos deadlines das tarefas. A segunda etapa corresponde a calcular a escala de execução das tarefas, ou seja, determinar que tarefa usa qual recurso a cada momento. Nem todas as abordagens propostas incluem um teste de escalonabilidade explícito. Entretanto, todas precisam gerar, em algum momento, uma escala de execução.

A figura 2.3 ilustra a classificação dos algoritmos que realizam testes de escalonabilidade presentes na literatura.

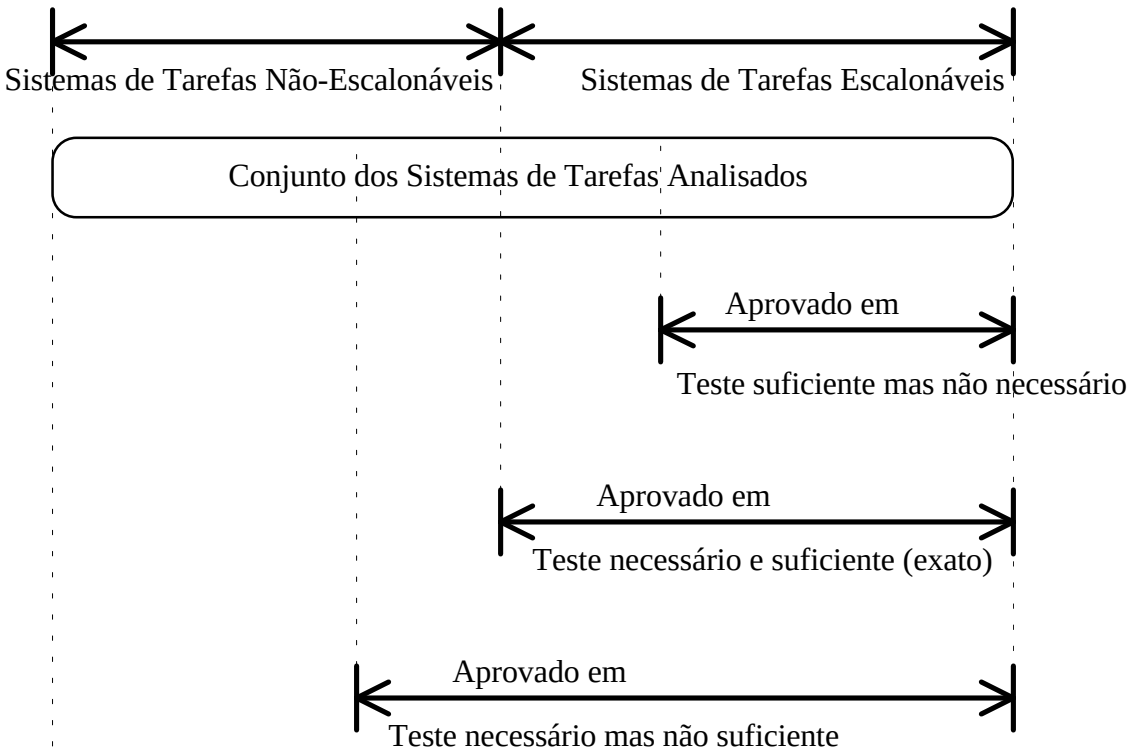

Figura 2.3 - Classificação dos teste de escalonabilidade.

Os algoritmos que realizam testes de escalonabilidade são classificados em:

- Suficiente mas não necessário: Todo conjunto de tarefas aprovado é escalonável, nada pode ser dito a respeito dos conjuntos reprovados;
- Necessário mas não suficiente: Todo conjunto de tarefas reprovado não é escalonável, nada pode ser dito a respeito dos conjuntos aprovados;
- Exato: Todo conjunto de tarefas aprovado é escalonável e todo conjunto reprovado não é escalonável.

Na maioria das vezes o teste de escalonabilidade é executado em tempo de projeto. Entretanto, em algumas soluções de escalonamento existe a necessidade de incluir novas tarefas no sistema em tempo de execução. Isto deve ser feito sem comprometer à garantia de escalonabilidade fornecida pelo teste de escalonabilidade executado em tempo de projeto. Neste caso, é executado um teste de escalonabilidade em tempo de execução para verificar se a nova tarefa pode ou não ser incluída no sistema. Testes de escalonabilidade empregados em tempo de execução são chamados de testes de aceitação.

Utilização do processador

Algumas soluções de escalonamento estão baseadas no conceito de utilização ("utilization") do processador. A utilização do processador $\mathbf{U_i}$, referente a tarefa $\mathbf{T_i}$, corresponde a fração do tempo total do processador que esta tarefa poderá ocupar, no pior caso. Considere uma aplicação composta por **N** tarefas. A utilização do processador $\mathbf{U_i}$, referente a tarefa $\mathbf{T_i}$, é definida como:

$\mathbf{U_i = C_i / P_i}$, para tarefas periódicas;
$\mathbf{U_i = C_i / I_i}$, para tarefas esporádicas.

Onde:
- $\mathbf{C_i}$ é o tempo máximo de computação da tarefa $\mathbf{T_i}$;
- $\mathbf{P_i}$ é o período da tarefa $\mathbf{T_i}$ (tarefas periódicas);
- $\mathbf{I_i}$ é o intervalo mínimo entre ativações da tarefa $\mathbf{T_i}$ (tarefas esporádicas).

A utilização total do processador **U**, devido ao conjunto de **N** tarefas, é definida como $U = \sum_{i=1,2,...,N} (Ui)$.

Carga computacional

Uma aplicação tempo real é constituída por um conjunto de tarefas. Para efeitos de escalonamento, o somatório destas tarefas constitui a carga ("task load") que a aplicação representa para os recursos do sistema computacional. Esta carga é dita limitada se todas as tarefas forem conhecidas em tempo de projeto, existir um número limitado de tarefas e todas as tarefas forem periódicas ou esporádicas.

A carga é ilimitada se existir ao menos uma tarefa aperiódica, cujo intervalo mínimo entre liberações seja zero. Neste caso, existe a possibilidade teórica (em função do modelo usado) de infinitas ativações de uma mesma tarefa no mesmo instante de tempo. A carga também será ilimitada quando o modelo admitir a criação dinâmica de tarefas. Para efeitos

de escalonamento, uma tarefa criada dinamicamente é normalmente tratada da mesma forma que uma tarefa aperiódica.

A carga que a aplicação representa para os recursos do sistema também pode ser classificada em estática ou dinâmica. A carga é definida como estática se permitir um tratamento de pior caso ainda em tempo de projeto. A carga será considerada dinâmica quando não permitir um tratamento de pior caso em tempo de projeto.

Para que a carga possa receber um tratamento de pior caso, ela deve ser limitada. Quando a carga é ilimitada, não é possível executar tal tratamento. Logo, é possível afirmar que os conceitos de carga limitada e carga estática são equivalentes. Da mesma forma, os conceitos de carga ilimitada e carga dinâmica também são equivalentes.

Escalonamento estático e Escalonamento dinâmico

Em [CHE 88] são definidos os conceitos de escalonamento estático e escalonamento dinâmico. Esta classificação é citada com frequência na bibliografia, como em [AUD 90c] e [KOP 92]. Cheng, Stankovic e Ramamritham definem em [CHE 88] escalonamento estático como aquele que "calcula a escala de execução das tarefas em tempo de projeto e requer completo conhecimento à priori das características das tarefas". Ainda citando [CHE 88], escalonamento dinâmico é aquele que "determina a escala de execução das tarefas em tempo de execução e permite que tarefas sejam dinamicamente invocadas". O diagrama abaixo resume a classificação proposta em [CHE 88].

		Conhecimento à priori da carga	
		SIM	NÃO
Cálculo da Escala de Execução	No projeto	*Esc. Estático*	*???*
	Na execução	*???*	*Esc. Dinâmico*

Existem propostas de escalonamento que exigem conhecimento à priori da carga ao mesmo tempo que calculam a escala de execução em tempo de execução. Tais propostas não podem ser enquadradas na classificação apresentada acima.

Neste trabalho, escalonamento estático será redefinido como aquele capaz de oferecer uma previsibilidade determinista em tempo de projeto. Para tanto, o escalonamento estático exige uma carga estática, limitada, conhecida em tempo de projeto. Por sua vez, o escalonamento será considerado dinâmico quando não oferecer uma previsibilidade determinista, sendo porém capaz de lidar com uma carga dinâmica, possivelmente ilimitada. Esta definição é compatível com a definição apresentada em [CHE 88], ao mesmo tempo que consegue classificar satisfatoriamente as propostas encontradas na bibliografia. O diagrama abaixo resume as definições adotadas neste trabalho. É importante observar que nada é dito com respeito ao momento do cálculo da escala de execução.

		Carga Estática/Limitada	
		SIM	NÃO
Garantia em Tempo de Projeto	SIM	*Esc. Estático*	*Impossível*
	NÃO	*Esc. Dinâmico*	*Esc. Dinâmico*

Na maioria das vezes, escalonamento estático aparece associado com carga estática e escalonamento dinâmico aparece associado com carga dinâmica. Entretanto, existe a possibilidade de um escalonamento dinâmico ser aplicado a uma carga estática. Isto significa não fornecer nenhuma garantia em tempo de projeto, mesmo a carga sendo estática e passível de uma análise de pior caso. Esta situação acontece quando os requisitos da aplicação não exigem tal garantia e o desejo é a otimização dos recursos. É possível construir um sistema mais barato, dimensionando-o para o caso médio e não para o pior caso. Ainda que a carga seja estática, a falta de um teste de escalonabilidade em tempo de projeto caracteriza o escalonamento como dinâmico.

Sistemas tempo real Hard e Soft

Uma classificação também muito utilizada na bibliografia divide os sistemas tempo real em dois tipos: HARD e SOFT. Um STR é dito HARD (STR-H) caso sua temporalidade seja crítica, ou seja, caso o dano causado por um deadline não cumprido seja maior que qualquer valor que pode ser obtido pelo correto funcionamento do sistema. Em outras palavras, falhas temporais em um STR-H apresentam consequências catastróficas, muito além de qualquer benefício obtido do sistema na ausência de falhas. Por outro lado, um STR é dito SOFT (STR-S) se o não cumprimento de um deadline diminui o benefício global do sistema mas não gera consequências catastróficas.

2.3 Classificação das Abordagens para o Escalonamento Tempo Real

Na literatura são identificadas diferentes propostas de como sistemas tempo real podem ser escalonados. Estas propostas muitas vezes diferem consideravelmente. Entretanto, alguns aspectos são relevantes para qualquer tentativa de classificação: a previsibilidade, a utilização de recursos e algoritmos de escalonamento empregados. A classificação das possíveis abordagens descrita nesta seção leva em consideração esses aspectos e é uma extensão da apresentada em [RAM 94].

A figura 2.4 mostra as 5 abordagens básicas para o escalonamento na construção de STR. Elas aparecem divididas em dois grupos, conforme o tipo de previsibilidade oferecida. O primeiro grupo de abordagens (garantia no projeto) é aquele capaz de oferecer uma previsibilidade determinista. É empregado em sistemas onde é necessário garantir, em tempo de projeto, que todas as tarefas serão executadas dentro de seus deadlines. Obviamente, esta garantia é obtida a partir de um conjunto de premissas, ou seja, uma determinada hipótese de carga ("load hypothesis") e uma hipótese de faltas ("fault hypothesis").

O escalonamento de um conjunto de tarefas é, muitas vezes, dividido em duas etapas: um teste de escalonabilidade que determina se é possível atender os deadlines e o cálculo da própria escala de execução das tarefas. O grupo "garantia no projeto" tem como característica, em suas duas abordagens, a execução "off-line" do teste de escalonabilidade.

Na primeira abordagem, executivo cíclico, além do teste, todo o trabalho de escalonamento é realizado em tempo de projeto ("off-line"). O resultado é uma grade ("time grid") que determina "qual tarefa executa quando em qual processador". Um exemplo desta abordagem é o sistema Mars ([DAM 89], [KOP 89]).

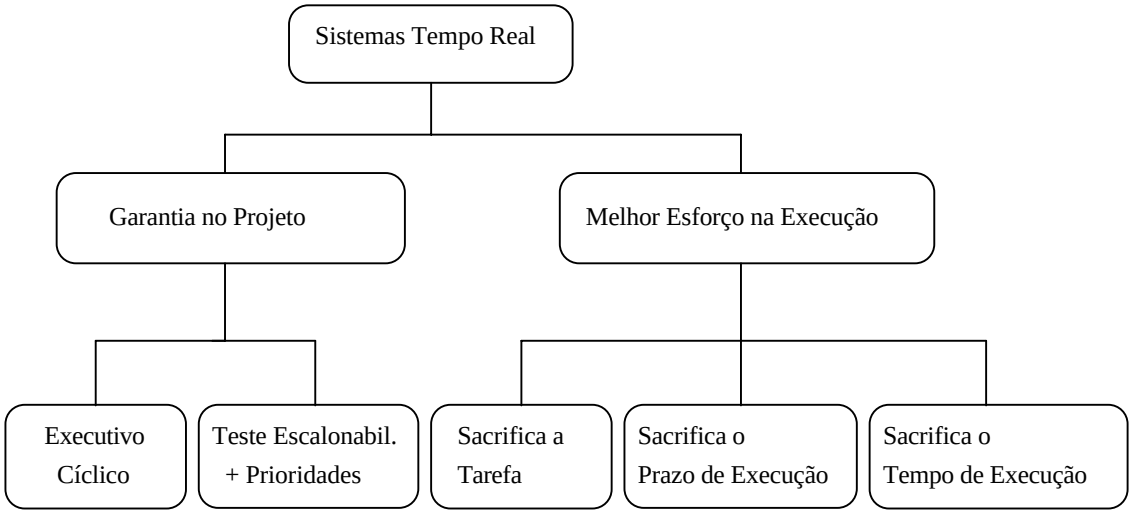

Figura 2.4 - Possíveis abordagens para sistemas tempo real.

Na outra abordagem (teste escalon.+prioridades) tarefas recebem uma prioridade e o teste de escalonabilidade é então executado determinando se existe a garantia de que todas as tarefas serão executadas dentro dos deadlines; isto tudo em tempo de projeto. Em tempo de execução, um escalonador preemptivo produz a escala de execução usando as prioridades das tarefas. Exemplos de propostas que seguem esta abordagem podem ser encontrados em [SHA 94], [AUD 93a] e [TIN 94a].

A garantia no projeto, ao oferecer uma previsibilidade determinista, implica na necessidade de uma reserva de recursos para o pior caso. Isto pode representar uma enorme subutilização de recursos. Por exemplo, o tempo médio de execução de muitos algoritmos é consideravelmente menor do que o tempo de execução no pior caso. Um outro problema com esta abordagem é a exigência de uma carga limitada e estática. Em resumo, é exigido o sacrificio de recursos e da flexibilidade do sistema com o objetivo de obter previsibilidade. Em [LOC 92] é feita uma comparação entre as abordagens deste primeiro grupo.

Nas abordagens do segundo grupo (melhor esforço na execução), não existe garantia, fornecida em tempo de projeto de que os deadlines serão cumpridos. O escalonamento tipo "melhor esforço" ("best effort") quando muito fornece uma previsibilidade probabilista sobre o comportamento temporal do sistema, a partir de uma estimativa da carga. Algumas propostas dentro desta linha oferecem uma "garantia dinâmica" ao determinar, em tempo de execução, quais prazos serão ou não atendidos.

Uma consequência imediata destas abordagens dinâmicas é a possibilidade de sobrecargas ("overload") no sistema. O sistema se encontra em estado de sobrecarga quando não é possível executar todas as tarefas dentro dos seus respectivos prazos. É importante observar que a sobrecarga não é um estado anormal, mas uma situação que ocorre naturalmente em sistemas que empregam uma abordagem tipo melhor esforço. Logo, é necessário um mecanismo para tratar a sobrecarga. As três abordagens identificadas segundo seus procedimentos de tratamento de sobrecarga são: descarte por completo de algumas tarefas ([RAM 89], [RAM 90]); execução de todas as tarefas com sacrifício do prazo de execução em algumas tarefas ([JEN 85]) e execução de todas as tarefas mas com sacrifício do tempo de execução de algumas tarefas ([LIU 94]).

A classificação das abordagens descrita acima considerou principalmente os aspectos carga, previsibilidade e momento do cálculo da escala de execução. A tabela abaixo lista todas as possíveis combinações para estes três aspectos e associa a cada combinação a respectiva abordagem (se existir).

Tipo de Carga	Garantia no Projeto (teste de escalonab.)	Escala Calculada	Abordagem
Limitada/Estática	Sim	Execução	*Teste + Prioridades*
Limitada/Estática	Sim	Projeto	*Executivo cíclico*
Limitada/Estática	Não	Execução	*Melhor esforço*
Limitada/Estática	Não	Projeto	*Possível, mas ... (1)*
Ilimitada/Dinâmica	Sim	Execução	*Impossível*
Ilimitada/Dinâmica	Sim	Projeto	*Impossível*
Ilimitada/Dinâmica	Não	Execução	*Melhor esforço*
Ilimitada/Dinâmica	Não	Projeto	*Impossível*

(1) Possível, mas sem sentido. Com carga estática e escala de execução calculada em tempo de projeto é possível conhecer completamente o comportamental temporal do sistema ainda em tempo de projeto. Uma garantia em tempo de projeto pode ser obtida através de uma simples inspeção da escala calculada.

É importante observar que prioridades podem ser utilizadas tanto em uma abordagem garantida quanto em uma abordagem melhor esforço. A garantia obtida em sistemas que trabalham com prioridades está associada com o teste de escalonabilidade. É claro que o desenvolvimento de um teste de escalonabilidade pode considerar uma forma específica de atribuir prioridades. Existem algoritmos para atribuir prioridades que são usados nos dois contextos. Por exemplo, o EDF ("earliest deadline first", descrito mais adiante neste capítulo) pode ser acompanhado de um teste de escalonabilidade em tempo de projeto, quando a carga é conhecida e limitada. Ele também é muito usado em abordagens tipo "sacrifica a tarefa", por apresentar propriedades interessantes, ainda que sem garantia em tempo de projeto.

2.3.1 Classe das Propostas com Garantia Baseadas em Executivo Cíclico

As propostas pertencentes a esta classe constroem, em tempo de projeto, uma grade que determina "qual tarefa executa quando". Qualquer conflito (recursos, precedência, etc) é resolvido durante a construção da grade. Em tempo de execução, um programa executivo simplemente dispara as tarefas no momento indicado pela grade, que é repetida indefinidamente. Neste tipo de proposta, é possível garantir que todos os deadlines serão cumpridos a partir de uma simples inspeção da grade gerada.

Em [XU 93a] é feito um levantamento abrangente dos algoritmos existentes dentro desta abordagem. O artigo descreve um modelo de tarefas de referência, composto por um conjunto de tarefas periódicas. Cada uma delas é caracterizada por período, deadline, tempo de computação no pior caso. Tarefas esporádicas são transformadas em periódicas. O modelo de referência comporta ainda relações de precedência e exclusão entre tarefas, além da existência de recursos que são disputados pelas tarefas (serialmente reusáveis). O artigo classifica diversas propostas encontradas na bibliografia, conforme as propriedades do modelo de referência que são suportadas e o tipo de algoritmo empregado.

Uma das propostas mais importantes dentro desta abordagem é o Sistema Mars ([DAM 89], [KOP 89]). A principal característica do sistema Mars ("Maintainable Real-Time System") é prover um desempenho previsível sob uma carga de pico e um cenário de faltas especificado. Em cada nível do sistema e nas ferramentas, as restrições temporais são garantidas de forma determinista. Foram concebidos para este fim uma arquitetura de hardware, um protocolo de comunicação, um sistema operacional, uma linguagem de programação, um modelo para a construção das aplicações, além de um conjunto de ferramentas para suportar desenvolvimentos de aplicações.

2.3.2 Classe das Propostas com Garantia Baseadas em Teste de Escalonabilidade e Prioridades

Nesta classe de propostas, as tarefas possuem prioridades. Em tempo de projeto, um teste de escalonabilidade avalia se existe ou não a possibilidade de algum deadline ser perdido. Este teste leva em consideração às propriedades temporais das tarefas. Em tempo de execução, um escalonador preemptivo escolhe sempre a tarefa pronta para executar com a prioridade mais alta. O importante nesta abordagem é manter a compatibilidade entre a forma como prioridades são associadas às tarefas e o teste de escalonabilidade empregado.

Em seu clássico artigo [LIU 73], Liu e Layland propuseram dois mecanismos de escalonamento tempo real baseados em prioridades. O algoritmo Taxa Monotônica (RM - "rate monotonic") trabalha com prioridades fixas e associa prioridades mais altas para as tarefas com menor período. O algoritmo Próximo Deadline (EDF - "earliest deadline first") trabalha com prioridades variáveis, executando antes a tarefa cujo deadline está mais próximo do momento atual.

Acompanhados de testes de escalonabilidade em tempo de projeto, estes algoritmos são capazes de fornecer uma previsibilidade determinista para o cumprimento de todos os deadlines das tarefas. É importante salientar que a garantia em tempo de projeto está associada com a execução de um teste de escalonabilidade apropriado. Os testes empregados são geralmente suficientes mas não necessários. Existem também alguns testes exatos. A forma como as prioridades são atribuidas às tarefas é, em geral, o ponto de partida para a construção destes testes. Durante a execução, é necessário apenas um escalonador preemptivo baseado em prioridades.

No artigo original, os modelos de tarefas eram bastante simples. A extensão do modelo de tarefas é limitada pela necessidade de refletir no teste de escalonabilidade qualquer alteração feita. Entretanto, nos últimos anos a abordagem baseada em prioridades ganhou importância. Diversos avanços na teoria aumentaram sua aplicabilidade. Também surgiram outros algoritmos além do EDF e do RM. Em [AUD 91b] pode ser encontrado um algoritmo ótimo para atribuir prioridades fixas em modelos de tarefas complexos.

A bibliografia disponível sobre Taxa Monotônica (RM - "rate monotonic") é bastante extensa. Um resumo do estado da arte pode ser encontrado em [SHA 94]. Revisões semelhantes também existem em [SHA 93] e [KLE 94]. Algumas extensões ao modelo básico do EDF podem ser encontradas na bibliografia. Em [CHE 90b] o modelo de tarefas do EDF passa a incluir recursos além do processador. Outras extensões ou análises podem ser encontradas em [LEH 89], [SHA 90], [BAK 91] e [JEF 92].

Deadline Monotônico (DM - "deadline monotonic") é uma proposta semelhante ao RM. Este método foi definido em [LEU 82] visando tarefas com deadline menor ou igual ao período. Uma prioridade fixa é associada a cada tarefa em tempo de projeto. Durante a execução é utilizado um escalonador preemptivo baseado em prioridades. Testes de escalonabilidade em tempo de projeto verificam se existe garantia no conjunto para os deadlines das tarefas. Em [AUD 90a] e [AUD 91a] são descritos testes de escalonabilidade para um modelo de tarefas onde o sistema computacional é composto por **N** tarefas que executam em um monoprocessador segundo a política do DM. As tarefas podem ser periódicas ou então tarefas esporádicas, com um intervalo de tempo mínimo entre ativações.

A análise baseada em períodos de ocupação[2] (BP - "busy-period analysis") permite encontrar o instante de conclusão de cada tarefa, no pior caso, quando prioridades fixas são empregadas. Se o instante de conclusão, no pior caso, for menor que o deadline da tarefa, significa que o deadline será sempre cumprido. Desta forma, esta análise pode ser usada como um teste de escalonabilidade em tempo de projeto. A análise BP admite modelos de tarefas mais flexíveis que aqueles suportados atualmente pelos testes de escalonabilidade definidos inicialmente para EDF, RM e DM. Entre outras coisas, deadlines podem ser menor, igual ou maior que o período das tarefas. Este tipo de análise apareceu antes em [LEH 90] e foi estendido em [TIN 94a].

Também existem na literatura abordagens que misturam prioridades fixas com prioridades variáveis. O objetivo destas propostas é juntar a maior eficiência dos algoritmos de prioridade variável com a facilidade de análise e a previsibilidade do comportamento em caso de falhas oferecidas pelos algoritmos baseados em prioridades fixas. Propostas deste tipo podem ser encontradas em [JEF 93] e [OLI 96a].

2.3.3 Classe das Propostas Melhor Esforço com Sacrifício de Tarefas

As propostas dentro desta classe não oferecem garantias em tempo de projeto. Isto permite que elas trabalhem com carga dinâmica e empreguem técnicas que determinam a ocupação dos recursos com eficiência. Toda vez que uma tarefa é ativada, o escalonador tenta posicioná-la na fila do processador de maneira a atender o prazo solicitado, sem comprometer as tarefas que já estão na fila. Se isto acontecer, esta ativação da tarefa terá obtido uma garantia em tempo de execução. Caso não seja possível garantir o deadline da tarefa, ela é descartada. Em geral, as tarefas que já obtiveram esta garantia em tempo de execução (foram inseridas na fila) não podem mais ser descartadas pelo escalonador. Algumas vezes, como em [RAM 94], esta abordagem é chamada de baseada em planejamento ("planning-based"). Em geral, os autores que trabalham seguindo esta abordagem alegam que os "níveis superiores da aplicação" deverão incluir um tratamento de exceção para quando tarefas são descartadas.

É importante observar que muitas propostas dentro desta abordagem procuram não excluir tarefas periódicas cuja escalonabilidade foi, de alguma forma, verificada em tempo de projeto. Isto acontece, por exemplo, com os algoritmos propostos em [RAM 89], [CHE 90a] e [ELH 94]. Entretanto, os algoritmos pertencentes a esta abordagem tratam basicamente do escalonamento de tarefas não garantidas em tempo de projeto. São

[2]Esta técnica também é conhecida por Análise Baseada em Janelas ("window-based analysis").

algoritmos que descartam a tarefa caso não seja possível cumprir seu deadline. Após o deadline, o valor da tarefa para o sistema cai a zero. Exemplos de propostas que seguem esta abordagem podem ser encontrados também em [SCH 92], [CHE 90a], [KOR 92] e [ELH 94]. Em [RAM 90] são apresentados dois algoritmos, baseados em heurísticas, para escalonar conjuntos de tarefas em multiprocessadores. O primeiro algoritmo é uma extensão de um algoritmo apresentado antes em [ZHA 87] para monoprocessadores.

2.3.4 Classe das Propostas Melhor Esforço com Sacrifício de Prazos

Para as abordagens apresentadas nas seções anteriores, o deadline é um instante no tempo após o qual a conclusão da tarefa não mais representa um benefício para o sistema. As propostas que seguem esta abordagem flexibilizam o conceito de deadline. Elas consideram que as tarefas representam diferentes níveis de benefício para o sistema, conforme o momento de sua conclusão. Desta forma, o conceito tradicional de deadline é uma simplificação desta visão mais ampla.

Um dos primeiros trabalhos seguindo esta abordagem pode ser encontrado em [JEN 85]. Os autores partem do conceito de que a conclusão de uma tarefa, ou de um conjunto de tarefas, contribui para o sistema com um benefício e o valor deste benefício pode ser expresso em função do instante de conclusão da tarefa (ou do conjunto).

2.3.5 Classe das Propostas Melhor Esforço com Sacrifício da Qualidade

Nesta abordagem, não é oferecido nenhum tipo de garantia em tempo de projeto. Em tempo de execução, as tarefas são escalonadas de forma a cumprirem seus deadlines. Em caso de sobrecarga, ao invés de descartar a tarefa ou comprometer o deadline, como nos casos anteriores, os algoritmos desta abordagem diminuem o tempo de execução da tarefa. Para isto, é necessário que cada tarefa possua opções do tipo "qualidade versus tempo de execução". Dependendo de como esta tarefa é programada, esta relação pode corresponder a uma curva contínua (refinamentos sucessivos ao longo da execução) ou a uma sequência de degraus (diferentes versões fornecendo diferentes níveis de qualidade).

Normalmente, as propostas que trabalham segundo esta abordagem não aparecem sozinhas, mas sim como componentes de uma abordagem mais ampla, a Computação Imprecisa ("imprecise computation") [LIU 94]. Computação Imprecisa será descrita na próxima seção.

2.4 Computação Imprecisa

Como colocado antes, os sistemas em geral, que não são do tipo tempo real, são caracterizados por uma abordagem do tipo "fazer o trabalho usando o tempo que for necessário". Já sistemas tempo real possuem uma abordagem diferente, pois o tempo é limitado. É preciso garantir que será possível atender aos prazos, geralmente impostos pelo ambiente do sistema. Logo, a preocupação é "garantir que o trabalho será concluído no tempo disponível".

A dificuldade encontrada no escalonamento tempo real levou alguns autores a proporem uma abordagem diferente, do tipo "fazer o trabalho possível dentro do tempo disponível". Isto significa sacrificar a qualidade dos resultados para poder cumprir os prazos

exigidos. Esta técnica, conhecida pelo nome de Computação Imprecisa, flexibiliza o escalonamento tempo real. As primeiras publicações a respeito datam de 1987 ([LIN 87a], [LIN 87b]). Elas descrevem trabalhos realizados na University of Illinois at Urbana-Champaign, por uma equipe integrada pelos professores Jane W. S. Liu e Kwei-Jay Lin, entre outros. É deles a maior parte dos artigos publicados sobre o assunto até hoje.

Computação Imprecisa está fundamentada na idéia de que cada tarefa do sistema possui uma parte obrigatória ("mandatory") e uma parte opcional ("optional"). A parte obrigatória da tarefa é capaz de gerar um resultado com a qualidade mínima, necessária para manter o sistema operando de maneira segura. A parte opcional então refina este resultado, até que ele alcançe a qualidade desejada. O resultado da parte obrigatória é dito impreciso ("imprecise result"), enquanto o resultado das partes obrigatória+opcional é dito preciso ("precise result"). Uma tarefa é chamada de tarefa imprecisa ("imprecise task") se for possível decompo-la em parte obrigatória e parte opcional.

Em situações normais, tando a parte obrigatória quanto a parte opcional são executadas. Desta forma, o sistema gera resultados com a precisão desejada. Se, por algum motivo, não for possível executar todas as tarefas do sistema, algumas partes opcionais serão deixadas de lado. Este mecanismo permite uma degradação controlada do sistema, na medida em que pode-se determinar o que não será executado em caso de sobrecarga.

É possível dizer que Computação Imprecisa faz uma composição das abordagens tipo melhor esforço (partes opcionais) com as abordagens que oferecem garantia (partes obrigatórias). A técnica como um todo é do tipo melhor esforço, pois não oferece uma previsibilidade determinista para todas as tarefas do sistema. Entretanto, as partes obrigatórias tomadas isoladamente formam um subproblema que deve ser tratado pelas abordagens que oferecem garantias. Neste subproblema devem ser observadas as condições necessárias para uma previsibilidade determinista, ou seja, carga limitada, conhecida e reserva de recurso para o pior caso.

As partes opcionais são executadas quando existirem recursos disponíveis, sem garantias em tempo de projeto. No máximo uma previsibilidade probabilista, se a carga for definida também em termos probabilistas. Isto permite a melhor utilização dos recursos pelas partes opcionais, que podem aproveitar recursos reservados em tempo de projeto e não utilizados pelas partes obrigatórias. Como não existe reserva de recursos para as partes opcionais, podem ocorrer sobrecargas temporárias. Neste caso, a qualidade do resultado é sacrificada para que os prazos sejam mantidos. Na falta de recursos, partes opcionais podem ser até totalmente descartadas.

Computação Imprecisa diferencia-se das demais abordagens tipo melhor esforço exatamente na forma como a sobrecarga é tratada. Nas demais abordagens do tipo melhor esforço, a sobrecarga é tratada através do simples descarte de tarefas ou da flexibilização do conceito de deadline. Na Computação Imprecisa, o tempo de computação das tarefas é negociado a partir da introdução do conceito de qualidade do resultado. Tarefas não são simplesmente descartadas, mas a qualidade do resultado produzido é parcialmente sacrificada. Isto gera uma redução na demanda pelos recursos do sistema que permite o atendimento dos deadlines, no seu sentido mais rigoroso.

O modelo de tarefas normalmente associado com Computação Imprecisa não exclui a existência de tarefas somente com parte obrigatória ou somente com parte opcional. Cabe

à semântica da aplicação definir quais são as partes obrigatórias e quais são as opcionais. Este é um modelo de tarefas mais flexível que o encontrado nas outras abordagens.

A aplicação da técnica Computação Imprecisa pode variar com relação a diversos aspectos. O restante deste capítulo apresenta uma visão geral da técnica. Os seguintes aspectos serão tratados ao longo do texto: motivação, formas de programação, função de erro, uso da função de erro no escalonamento e soluções para o problema do escalonamento. Não serão discutidos aspectos ligados à organização do software.

2.4.1 Motivações

Diversas motivações para o emprego de Computação Imprecisa são citadas na bibliografia. Cada uma delas emprega a técnica de uma forma diferente, para atender a diferentes objetivos de projeto. Esta seção resume as propostas encontradas na bibliografia com respeito ao aspecto "motivação".

Em sistemas computacionais cuja carga é dinâmica, Computação Imprecisa representa um mecanismo para tratamento de sobrecarga que respeita os deadlines das tarefas. É um tratamento diferente da simples rejeição da tarefa ou da flexibilização do conceito de deadline. O escalonamento é feito no sentido de obter o melhor comportamento possível para o sistema, dadas as restrições de recursos e deadlines.

Em uma situação de carga estática, sempre é possível reservar recursos para o pior caso de todas as tarefas e garantir todos os deadlines. Entretanto, em sistemas grandes e complexos, o custo de um sistema com tal nível de garantia pode ser proibitivo. Uma abordagem tipo melhor esforço é justificada neste contexto pelo aspecto econômico. Através do emprego da Computação Imprecisa, o custo do sistema diminui, pois são reservados recursos apenas para as partes obrigatórias das tarefas.

Computação Imprecisa também pode ser usada para viabilizar o emprego, em sistemas de tempo real, de algoritmos cujo tempo de execução no pior caso torna o escalonamento inviável. Alguns autores acreditam que este pode ser o caminho para obter-se uma previsibilidade determinista em ambientes não deterministas, que requerem algoritmos complexos, cujo tempo de execução no pior caso é difícil de prever. Um exemplo de aplicação que emprega Computação Imprecisa com esta finalidade pode ser encontrado em [KOP 89]. Também em [LIU 94] é reconhecido que "na prática, é frequentemente impossível calcular um limite para o tempo necessário para produzir um resultado aceitável". É citado o exemplo de uma tarefa que, interativamente, procura a raiz de um polinômio.

Liu e outros descrevem Computação Imprecisa em [LIU 94] como um mecanismo para tratar de sobrecargas temporárias em sistemas cuja carga é dinâmica. Desta forma, tais sistemas ficariam mais robustos e confiáveis. Como exemplos de aplicação, são citados o processamento de imagens e o rastreamento de objetos. Também é sugerido em [LIU 94] o uso de Computação Imprecisa como forma de tolerância a faltas em sistemas tempo real. A execução da parte obrigatória da tarefa pode ser vista como uma recuperação em avanço ("forward recovery"), quando uma falta qualquer impede a execução completa da parte opcional da tarefa. Resultados usáveis, ainda que imprecisos, aumentam a disponibilidade do sistema.

Uma linha de raciocínio semelhante é seguida em [YU 92] e [SHI 94]. Computação Imprecisa é utilizada em [YU 92] para que o sistema possa tolerar a perda de recursos computacionais em função de falhas no hardware. Na medida em que alguns processadores falham, uma mudança no modo de operação aumenta a carga nos processadores que continuam funcionando. Uma degradação suave acontece na medida em que resultados menos precisos são gerados em função desta sobrecarga. Também em [SHI 94], Computação Imprecisa é citada como uma técnica que poderia prover tolerância a faltas em sistemas tempo real. Desta forma, a presença de elementos faltosos em um sistema tempo real resulta em uma redução temporária na qualidade dos serviços prestados, com o objetivo de permitir ao sistema continuar atendendo os prazos das tarefas críticas.

Em [GAR 94] é feita uma retrospectiva dos trabalhos na área de inteligência artifical visando aplicações em tempo real. Garvey e Lesser citam Computação Imprecisa como a forma natural para implementar algoritmos tipo qualquer-tempo ("anytime algorithms"). Um algoritmo tipo qualquer-tempo é formado por refinamentos iterativos que, a qualquer momento, podem ser interrompidos e a melhor resposta até o momento é fornecida. É esperado que a qualidade da resposta melhore na medida em que o algoritmo tiver mais tempo para executar.

2.4.2 Formas de Programação

Existem 3 formas básicas de programar usando Computação Imprecisa normalmente citadas na literatura: a programação pode ser feita com funções monotônicas, funções de melhoramento ou múltiplas versões.

As *funções monotônicas* ("monotone functions") são aquelas cuja qualidade do resultado aumenta (ou pelo menos não diminui) na medida em que o tempo de execução da função aumenta. As computações necessárias para obter-se um nível mínimo de qualidade correspondem à parte obrigatória. Qualquer computação além desta será incluída como parte opcional. O nível mínimo de qualidade deve garantir uma operação segura do sistema, enquanto a parte opcional refina progressivamente o resultado da tarefa. Segundo [LIU 91], algoritmos deste tipo podem ser encontrados nas áreas de cálculo numérico, estimativa probabilista, pesquisa heurística, ordenação e consulta a banco de dados. Este tipo de função faz com que o escalonador tenha que decidir quanto tempo de processador cada parte opcional deve receber, visando o benefício do sistema como um todo. É a forma de programação que fornece maior flexibilidade ao escalonador.

Funções de melhoramento ("sieve functions") são aquelas cuja finalidade é produzir saídas no mínimo tão precisas quanto as correspondentes entradas. O valor de entrada é melhorado de alguma forma. Esta melhoria pode ser na dimensão tempo, na dimensão valor ou em ambas. Se o resultado recebido como entrada por uma função de melhoramento é aceitável como saída, então a função pode ser completamente omitida (não executada). As funções de melhoramento normalmente formam partes opcionais que seguem algum cálculo obrigatório. Tipicamente, não existe benefício em executar uma função de melhoramento parcialmente. Isto significa que o escalonador deve optar, antes de iniciar a tarefa, em executa-la completamente ou não executa-la. Um exemplo citado em [LIU 91] é o processamento de sinais de radar. Nestes, o passo que computa uma nova estimativa para o nível de ruído no sinal recebido pode ser omitido e a estimativa anterior usada no lugar. Também algoritmos de processamento de imagens são capazes de produzir imagens

razoáveis mesmo antes de terminar. Outros exemplos são métodos numéricos que fazem aproximações sucessivas e pesquisas heurísticas em árvore.

Uma tarefa também pode ser implementada através de *múltiplas versões* ("multiple versions"). Normalmente são empregadas duas versões. A versão primária gera um resultado preciso, porém possui um tempo de execução no pior caso desconhecido ou muito grande. A versão secundária gera um resultado impreciso, porém seguro para o sistema, em um tempo de execução menor e bem conhecido. A cada ativação da tarefa, cabe ao escalonador escolher qual versão será executada. No mínimo, deve ser garantido tempo de processador para a execução da versão secundária. Esta técnica, usada na aplicação exemplo descrita em [KOP 89], é a mais flexível do ponto de vista da programação.

Uma discussão sobre como programar tarefas imprecisas usando a linguagem ADA podem ser encontrada em [LIU 88] e [AUD 91c]. Em [KEN 91] é descrita a linguagem Flex, com suporte específico para este tipo de programação.

2.4.3 Função de Erro

Quando a parte opcional de uma tarefa é executada completamente, ela gera um resultado com qualidade máxima. Porém, quando esta mesma parte opcional não é executada completamente, o resultado gerado possui uma qualidade inferior. Para efeitos de escalonamento, muitas propostas associam um valor de erro a cada parte opcional não executada completamente. Este valor de erro quantifica a diferença entre a qualidade do resultado preciso e a qualidade do resultado efetivamente gerado. É suposto que os erros associados com tarefas individuais contribuem, de alguma forma, para a redução da qualidade do sistema como um todo.

Na literatura também pode ser encontrado o conceito de benefício gerado pela parte opcional. O benefício é definido com um sentido oposto ao do erro. Em outras palavras, uma parte opcional executada completamente gera um erro zero e um benefício máximo. Uma parte opcional completamente descartada gera um erro máximo e um benefício zero. Neste texto, serão usados os dois termos, conforme a conveniência do momento.

É preciso definir como calcular este valor de erro. Na maioria das propostas encontradas na literatura, este erro é suposto proporcional ao tempo de execução que faltou para concluir a parte opcional em questão. Em outras palavras, é suposta uma reta para a relação "erro da parte opcional" versus "tempo de execução", como ilustra o gráfico na figura 2.5.

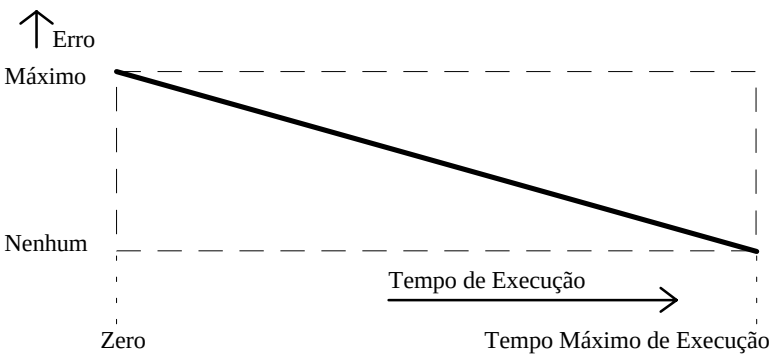

Figura 2.5 - Função de erro na forma de uma reta.

É reconhecido que, na prática, a função de erro não tem sempre o formato de uma reta. Dependendo do algoritmo em questão, esta curva pode ser linear, côncava, convexa ou ainda possuir um formato mais complexo. Por exemplo, um algoritmo para cálculo numérico que realiza iterações que convergem para o resultado possivelmente possui uma curva côncava, como ilustra a figura 2.6. No início da execução do algoritmo, o erro do resultado diminui rapidamente, pois cada iteração representa um salto em direção ao resultado final. No final, com o resultado impreciso próximo do resultado final, cada iteração faz apenas um pequeno refinamento. Existem na bibliografia muito poucos trabalhos que não utilizam uma reta como função de erro.

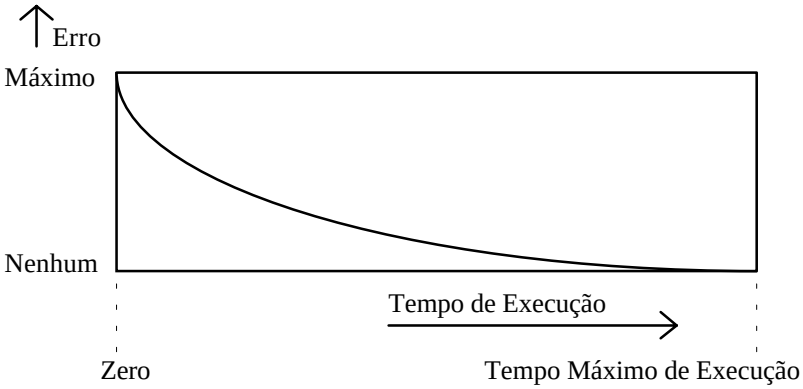

Figura 2.6 - Função de erro na forma de uma curva côncava.

Em [LIU 94] é proposto que o erro associado com uma parte opcional seja função do seu tempo de execução e também da qualidade dos seus dados de entrada. É normal que os dados computados por uma tarefa sejam os dados de entrada em uma tarefa sucessora. Neste caso, se a qualidade destes dados for muito baixa, a tarefa que os recebe pouco poderá fazer. A figura 2.7 ilustra a situação onde uma função de melhoramento perde sua capacidade de gerar benefício para o sistema se os dados de entrada forem muito ruins ou muito bons.

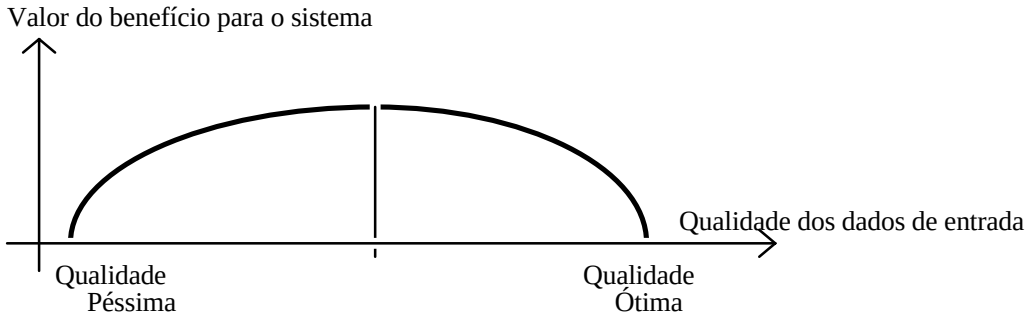

Figura 2.7 - Benefício gerado em função dos dados de entrada.

A figura 2.8 ilustra a combinação de tempo de execução e qualidade dos dados de entrada como determinantes do benefício gerado pela parte opcional. O resultado é um conjunto de retas. Uma outra possibilidade é descrita em [FEN 94], onde a qualidade dos

dados de entrada não afeta diretamente a qualidade final mas sim o tempo de execução da tarefa. Neste caso, dados de entrada com melhor qualidade resultam em um tempo de execução menor.

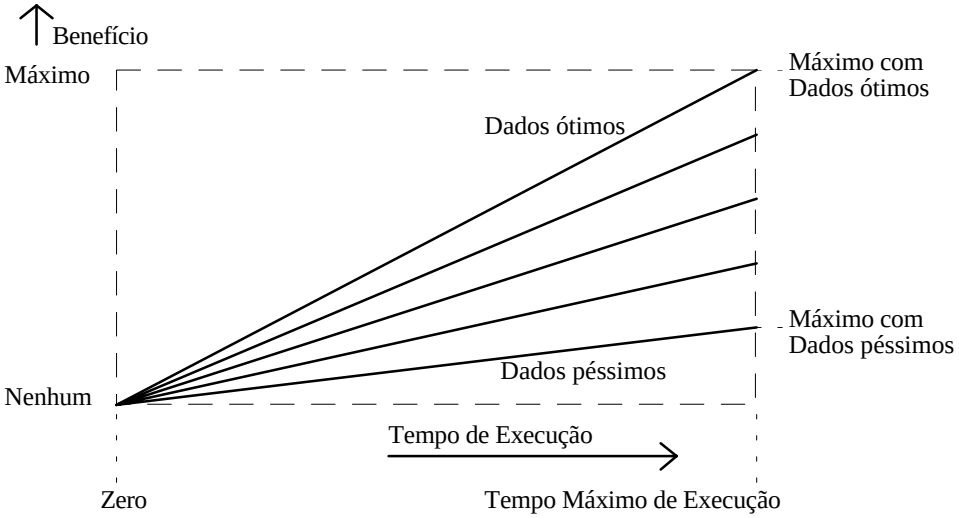

Figura 2.8 - Função de erro na forma de um conjunto de retas.

2.4.4 Uso da Função de Erro no Escalonamento

Toda proposta, dentro da Computação Imprecisa, necessita explicitar qual o objetivo a ser considerado no escalonamento das partes opcionais. Este objetivo será utilizado em caso de sobrecarga. Quando não é possível executar completamente todas as partes opcionais, é necessário algum critério para escolher quais partes opcionais terão seu tempo de execução sacrificado (parcialmente ou totalmente). Normalmente, este critério é definido a partir de uma medida do erro introduzido no sistema pelas tarefas tomadas individualmente. Esta medida é a função de erro, descrita na seção anterior.

Existem na bibliografia diversas propostas para o uso da função de erro no escalonamento. Muitas vezes, os objetivos encontrados foram escolhidos mais pela sua elegância matemática do que a partir de um estudo aprofundado dos requisitos das aplicações tempo real. Não existem trabalhos na bibliografia discutindo, sob a ótica da engenharia de software, a utilidade ou aplicabilidade das funções de erro propostas e a forma como são empregadas no escalonamento das partes opcionais.

Abaixo estão os objetivos normalmente encontrados na literatura com relação ao uso da função de erro no escalonamento:

Minimiza Erro Total

Supõe que cada parte opcional não executada completamente contribui com um valor de erro para o sistema como um todo. Procura minimizar o erro total do sistema, definido como o somatório dos erros associados com as partes opcionais não executadas. Algumas propostas associam diferentes pesos às tarefas, os quais são multiplicados ao valor do erro no momento do somatório.

Minimiza Erro Individual Máximo

Associa com cada parte opcional não executada completamente um valor de erro. Procura minimizar o maior erro observado no sistema, considerando os erros gerados em cada ativação de tarefa individualmente.

Minimiza Erro Total Após Minimizar Erro Individual Máximo

Minimiza o erro total do sistema, após ter conseguido minimizar o erro individual máximo. Erro total e erro individual máximo como definidos acima. Entre todas as soluções de escalonamento que minimizam o erro individual máximo, escolhe aquela que obtém o menor erro total.

Minimiza Erro Individual Máximo Após Minimizar Erro Total

Minimiza o erro individual máximo do sistema, após ter conseguido minimizar o erro total. Erro total e erro individual máximo como definidos acima. Entre todas as soluções de escalonamento que minimizam o erro total, escolhe aquela que obtém o menor erro individual máximo.

Minimiza Número de Partes Opcionais Descartadas

Minimiza o número de parte opcionais não executadas completamente. Usada normalmente em modelos de tarefas que não admitem a execução parcial de uma parte opcional. Cada parte opcional é executada completamente ou descartada completamente.

Minimiza Erro Médio do Sistema (tarefas periódicas)

Usado em sistemas com tarefas periódicas. Cada liberação da tarefa implica na liberação de sua parte opcional, que portanto também é periódica. A cada parte opcional não executada completamente é associado um valor de erro. Procura minimizar o erro médio do sistema, definido como o somatório dos erros médios associados com cada tarefa.

Minimiza Erro Individual Acumulado Máximo (tarefas periódicas)

Também usado em sistemas com tarefas periódicas. A cada parte opcional não executada completamente é associado um valor de erro. O erro associado com cada parte opcional em particular é acumulado (somado) pela tarefa até que a sua parte opcional seja completamente executada. Quando a sua parte opcional é completamente executada, o erro acumulado pela tarefa é zerado. O objetivo é minimizar o erro máximo acumulado por qualquer tarefa do sistema, observado a qualquer momento.

Escalona Conforme a Importância

Existem propostas que não associam funções de erro às tarefas. Simplesmente cada tarefa possui uma medida de importância relativa. Durante a execução do sistema, o objetivo é sacrificar antes as partes opcionais das tarefas de menor importância.

2.4.5 Escalonamento

Existem na bibliografia diversas propostas usando técnicas identificadas como Computação Imprecisa. Cada proposta define as propriedades do modelo de tarefas considerado, define o conceito de escalonamento ótimo e propõe algoritmos de escalonamento apropriados. A partir de variações nas propriedades das tarefas, na definição da função de erro e na forma como a função de erro é usada no escalonamento, é possível criar inúmeros problemas de escalonamento tempo real para Computação Imprecisa.

A análise matemática dos algoritmos de escalonamento requer a formalização dos conceitos de tarefa, parte obrigatória, parte opcional, prazo, benefício, etc. A maioria das propostas modelam cada tarefa T_i como uma composição de uma parte obrigatória M_i e uma parte opcional O_i. Existe uma relação de precedência entre M_i e O_i. Cada tarefa T_i é caracterizada por um momento de pronto ("ready time"), deadline, tempo de execução ("processing time") e importância para o sistema ("weight"). O tempo de pronto e o deadline de M_i e O_i são os mesmos de T_i. Os tempos de execução de M_i e O_i somados resultam no tempo de execução de T_i. Nesta formalização é necessário uma estimativa para o tempo de execução de O_i, mesmo que este seja exageradamente grande. As formas básicas de programação são modeladas da seguinte maneira:

- Função Monotônica: M_i representa as computações necessárias para obter-se um resultado de qualidade mínima (parte obrigatória) e O_i representa a melhoria deste resultado (parte opcional).
- Função de Melhoramento: Uma função de melhoramento obrigatória deve ser representada por M_i, enquanto uma função de melhoramento opcional aparece como O_i.
- Múltiplas Versões: M_i representa a execução da versão secundária, enquanto M_i+O_i representa a execução da versão primária.

Um dos principais aspectos da Computação Imprecisa é o escalonamento das tarefas. A parte obrigatória de uma tarefa deve ter seu deadline garantido pelo escalonador. Para tanto, podem ser empregados, em tempo de projeto, algoritmos que trabalham com o pior caso e fornecem uma previsibilidade determinista, tais como o Rate-Monotonic ([LIU 73]) e o Deadline-Monotonic ([AUD 90a]). Já as partes opcionais são escalonadas (ou não) visando maximizar a utilidade do sistema em tempo de execução ou, de forma equivalente, minimizar o erro gerado pela não execução de algumas computações opcionais.

Usualmente, o termo escalonamento preciso ("precise schedule") é usado para descrever uma solução de escalonamento que executa todas as tarefas completamente, inclusive a parte opcional. Desta forma, todas as tarefas sempre geram um resultado preciso. Já um escalonamento viável ("feasible schedule") é aquele onde as tarefas sempre executam completamente sua parte obrigatória, mas as partes opcionais podem ou não ser executadas. Obviamente, todo escalonamento preciso é também um escalonamento viável, mas o contrário não é sempre verdadeiro.

Em qualquer formalização de tarefa imprecisa, a forma de programação empregada interfere com o comportamento desejado para o escalonador. Quando uma função

monotônica é empregada, o tempo de processador alocado à parte opcional pode estar entre zero e o seu tempo total de execução. Isto porque, em uma função monotônica, qualquer tempo de processador fornecido ajuda a melhorar a qualidade do resultado. Quando a parte opcional executa uma função de melhoramento, não existe benefício em executa-la parcialmente. Assim, o escalonador deve decidir se executa a função de melhoramento completamente ou a descarta. O mesmo vale para múltiplas funções, onde não existe sentido em executar a função primária parcialmente. Esta característica das duas últimas formas de programação é chamada, na bibliografia, de restrição tipo 0/1 ("0/1 constraint"). Ela restringe de certo modo a flexibilidade do escalonador.

2.4.5.1 Classificação das Propostas

Cada proposta envolvendo Computação Imprecisa é caracterizada pelo:
- Modelo de tarefas considerado;
- Função de erro e seu uso no escalonamento;
- Algoritmos de escalonamento propostos.

Uma das classificações possíveis para as propostas encontradas na literatura é quanto ao modelo de carga suposto. Com respeito a este aspecto, existem três tipos de propostas, como ilustra a figura 2.9. É importante destacar que esta taxonomia considera apenas a forma como as propostas existentes na literatura abordam o problema. Em particular, como as propostas existentes na literatura abordam a questão da carga.

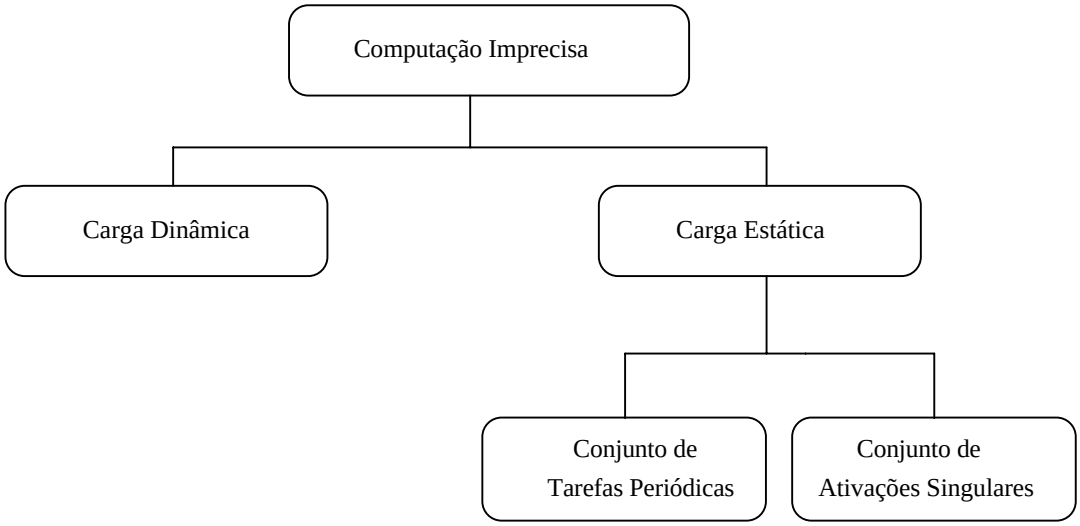

Figura 2.9 - Classificação das Propostas que empregam Computação Imprecisa.

Existem propostas que definem como carga do sistema um conjunto dinâmico de ativações de tarefas. Neste modelo de carga, tarefas imprecisas são dinamicamente ativadas, durante a execução do sistema. Em tempo de execução, o algoritmo de escalonamento procura aumentar o benefício gerado pelo sistema da melhor maneira possível. Como esta carga é potencialmente ilimitada, este algoritmo sozinho não consegue garantir que as partes obrigatórias de todas as tarefas serão executadas antes do respectivo deadline. Em função disto, as propostas que seguem esta linha consideram que as tarefas apresentadas ao escalonador satisfazem à restrição de que as partes obrigatórias são sempre escalonáveis ("feasible mandatory constraint"). Em outras palavras, é suposto que sempre é possível obter um escalonamento viável.

Existem dois caminhos possíveis para satisfazer à restrição de que as partes obrigatórias são sempre escalonáveis:

- A carga representada apenas pelas partes obrigatórias não nulas é, na realidade, limitada e conhecida antes da execução do sistema. Em tempo de projeto, um teste de escalonabilidade fornece a garantia de que todas as partes obrigatórias serão concluídas antes do respectivo deadline. Em tempo de execução, o escalonador trabalha como se a carga fosse completamente dinâmica, apresentando um comportamento que não compromete o resultado do teste de escalonabilidade.

- O escalonador rejeita tarefas imprecisas ativadas dinamicamente cuja parte obrigatória não pode ser executada dentro do deadline. Neste caso, a aplicação é obrigada a realizar algum tipo de tratamento de exceção em função das tarefas rejeitadas.

Algumas motivações para o uso de Computação Imprecisa com carga estática estão citadas no ítem 2.4.1. As propostas existentes na bibliografia que trabalham com carga estática diferenciam-se na forma como a carga é descrita. Elas formam duas subclasses de propostas.

Existem propostas que consideram como carga do sistema um conjunto estático de tarefas periódicas. Neste caso, a carga é limitada e conhecida, permitindo que todas as partes obrigatórias sejam garantidas em tempo de projeto. As propostas desta linha diferem quanto ao momento de decidir qual será o tempo de execução de cada parte opcional. Normalmente, o escalonador decide o quanto executar de cada parte opcional em tempo de execução, ativação por ativação, conforme a disponibilidade de recursos no momento. Algumas propostas são muito mais restritas e decidem o quanto executar de cada parte opcional ainda em projeto. Neste caso, o erro associado com cada parte opcional já é conhecido antes da execução e a técnica de Computação Imprecisa é aplicada apenas em tempo de projeto.

Finalmente, algumas propostas mais antigas consideram que a carga do sistema é formada por um conjunto estático de ativações singulares (individuais), completamente conhecido antes da sua execução. Desta forma, em tempo de projeto é possível calcular um escalonamento onde as partes obrigatórias são completamente executadas antes do respectivo deadline e o objetivo com respeito às partes opcionais é atendido. O resultado deste calculo é uma escala de execução, ou seja, um escalonamento que indica "qual tarefa ocupa qual processador quando" e será executada uma única vez. Esta é uma situação diferente daquela encontrada no executivo cíclico, onde as tarefas são periódicas e a grade é executada periodicamente. Este grupo de propostas possui pouco valor prático por si só e não houve novas contribuições nos últimos anos. Seu mérito foi servir como base para o grupo de propostas que lidam com carga dinâmica.

Independentemente do modelo de carga adotado, existem muitas outras variações com respeito às propriedades do modelo de tarefas. Entre as mais importantes estão:

- Tarefas podem ou não possuir restrição do tipo 0/1;
- Tarefas podem ou não possuir pesos diferentes;
- Tarefas podem ou não possuir relações de precedência;
- Tarefas podem ou não possuir relações de exclusão mútua.

As seções 2.4.3 e 2.4.4 trataram da função de erro e seu uso durante o escalonamento. Na verdade, é possível imaginar dezenas de variações com respeito a estes aspectos. Muitas formas de uso das funções de erro podem ser aplicadas junto com qualquer modelo de carga (por exemplo, minimizar erro total). Entretanto, algumas formas de uso das funções de erro somente fazem sentido em determinado modelo de carga (por exemplo, minimizar o erro máximo acumulado por tarefas periódicas).

2.4.5.2 Propostas com Carga Dinâmica

Nesta seção serão apresentadas propostas que trabalham com carga dinâmica. Em outras palavras, o escalonador deve ser capaz de lidar com tarefas conhecidas somente durante a execução. Todas as propostas dentro desta classe trabalham a partir da suposição que partes obrigatórias são sempre escalonáveis ("feasible mandatory constraint"). É suposto que no momento da ativação de uma nova tarefa, sua parte obrigatória e as porções das partes obrigatórias ainda não concluídas podem ser escalonadas satisfatoriamente ("feasibly scheduled"), para concluir antes de seus respectivos deadlines.

Minimiza Erro Total [SHI 92]

Em [SHI 92] são apresentados três algoritmos para escalonar conjuntos dinâmicos de tarefas imprecisas de forma preemptiva em monoprocessador. Tarefas são conhecidas pelo escalonador em tempo de execução ("on-line"). Os algoritmos são aplicáveis a modelos de tarefas levemente diferenciados com respeito à carga. Entretanto, o modelo de tarefas básico é o mesmo para os três algoritmos. Cada tarefa T_i, com **i** entre **1** e **N**, é caracterizada por um momento de chegada L_i ("arrival time", quando o escalonador toma conhecimento da tarefa), por um momento de pronto R_i ("ready time", quando a tarefa pode iniciar a execução), um deadline D_i e um tempo total de processamento C_i.

Minimiza Erro Total ou Número de Partes Opcionais Descartadas [HO 92a]

Em [HO 92a] é discutido o escalonamento preemptivo de tarefas imprecisas em monoprocessador. As partes opcionais possuem restrição tipo 0/1, ou seja, devem ser executadas totalmente para representarem algum benefício ao sistema. Esta restrição surge quando a Computação Imprecisa é programada através de função de melhoramento ou múltiplas versões. Esta restrição no modelo de tarefas não impede a programação através de funções monotônicas, apenas reduz as alternativas do escalonador.

Minimiza Erro Total e Erro Individual Máximo [HO 94] e [HO 92b]

Em [HO 94] são apresentados dois algoritmos para escalonar um conjunto de **N** tarefas imprecisas em multiprocessador com **V** processadores idênticos. As tarefas consideradas são indepedentes e o escalonamento é preemptivo. Os dois algoritmos diferem com respeito ao objetivo do escalonamento. É assumido que o conjunto de tarefas é sempre viável, no sentido de que as partes obrigatórias sempre podem ser escalonadas para serem concluídas antes do deadline.

2.4.5.3 Propostas com Carga Estática, Conjunto de Tarefas Periódicas

Nesta seção serão apresentadas propostas que supõe como carga um conjunto estático de tarefas periódicas, conhecidas em tempo de projeto.

Escalona Conforme Importância [AUD 91c]

Em [AUD 91c], o escalonamento é feito a partir do conceito de tarefas prólogo, epílogo e opcionais. Cada tarefa recebe uma prioridade fixa diferente. Um teste de escalonabilidade é usado para verificar se os prazos das tarefas obrigatórias (prólogos e epílogos) serão cumpridos. Tarefas são cíclicas com período conhecido ou esporádicas com intervalo mínimo entre ativações. Cada tarefa possui uma deadline, menor ou igual ao período. No caso das tarefas obrigatórias, o tempo de execução no pior caso é conhecido. As tarefas opcionais são ignoradas pelo teste de escalonabilidade em tempo de projeto, pois seus deadlines não são garantidas. É suposto no modelo um processador funcionando com escalonador preemptivo.

Minimiza Erro Médio Não Acumulativo [CHU 90]

Em [CHU 90] é discutido o problema de escalonar tarefas imprecisas que são periódicas. Aplicações deste tipo são classificadas como possuindo tarefas tipo N-AC ou tarefas tipo AC.

Uma tarefa é do tipo N-AC quando a sua parte opcional é realmente opcional, no sentido de que poderá não ser executada jamais. A qualidade do resultado gerado por uma tarefa tipo N-AC é medida em termos do erro médio da tarefa percebido ao longo de vários períodos consecutivos. Somente o efeito médio dos erros é observável e relevante. Um exemplo desta situação, citado em [CHU 90], pode ser encontrado em tarefas que recebem, melhoram ("enhance") e transmitem quadros ("frames") de imagens de vídeo. Embora a tarefa seja executada periodicamente, os erros em resultados gerados em diferentes ativações são independentes entre si.

Minimiza Erro Acumulado Máximo [CHU 90]

Ainda considerando as propostas em [CHU 90], se a tarefa é do tipo AC, o erro percebido em diferentes períodos (liberações) da tarefa possuem um efeito cumulativo. Isto torna necessário gerar um resultado preciso (completar a parte opcional) em algum período entre vários períodos consecutivos. Por exemplo, considere uma tarefa (descrita em [CHU 90]) que processa o sinal de radar retornado por um alvo e gera um comando para controlar o movimento do alvo. Quando a tarefa não é completamente executada, ela gera uma estimativa grosseira da posição. É necessário que, ocasionalmente, a tarefa seja executada completamente. Uma sequência de resultados imprecisos pode fazer o sistema perder o alvo de vista.

Minimiza Erro Total em Tempo de Projeto [YU 92]

Em [YU 92] é apresentada uma solução para o problema de alocação e escalonamento de tarefas imprecisas replicadas em um multiprocessador. Em tempo de projeto, heurísticas são empregadas para alocar as réplicas aos processadores e definir qual deverá ser o tempo de processador alocado para cada uma. Esta flexibilidade na determinação do tempo de execução das tarefas está ligada à existência de uma parte opcional em cada tarefa. Localmente é empregado EDF ("Earliest Deadline First") ou RM ("Rate-Monotonic") para escalonar cada processador.

2.4.5.4 Propostas com Carga Estática, Conjunto de Ativações Singulares

Nesta seção serão apresentadas propostas que supõe como carga um conjunto estático de ativações individuais, conhecido antes da execução. São propostas com um

modelo de carga muito restrito. Elas serão descritas por uma questão de completude do levantamento.

Minimiza Erro Total [LIU 91]

Em [LIU 91] são apresentados dois algoritmo para escalonar conjuntos de **N** tarefas imprecisas. Cada tarefa T_i, com **i** entre **1** e **N**, é caracterizada por um momento de pronto R_i ("release time"), uma deadline D_i, um tempo total de processamento C_i e um peso W_i. O modelo de tarefas adotado permite relações de precedência entre as tarefas do conjunto.

Minimiza Erro Total [SHI 89]

Em [SHI 89] é apresentado um algoritmo para escalonar conjuntos de **N** tarefas imprecisas em um monoprocessador. O modelo de tarefas é o mesmo da proposta apresentada em [LIU 91].

2.4.6 Computação Imprecisa em Ambiente Distribuído

Praticamente inexiste na literatura trabalhos analisando o emprego de Computação Imprecisa em ambiente distribuído. Em [KOP 89] aparece um dos poucos empregos de Computação Imprecisa neste contexto. A mesma tarefa possui uma versão primária e uma versão secundária. A versão primária fornece resultado com qualidade máxima, mas não é garantida. A versão secundária fornece resultado minimamente aceitável, mas é garantida. As duas versões executam paralelamente em processadores diferentes. A cada ativação, se a versão primária termina dentro do prazo, seus resultados são utilizados. Caso contrário, são utilizados os resultados da versão secundária, a qual é garantido que termina sempre dentro do prazo.

Em [HUA 95] o conceito de Computação Imprecisa é empregado em sistemas distribuídos como um mecanismo para controlar congestionamentos na transmissão de vídeo em tempo real. Aplicação semelhante é descrita em [FEN 96] para a transmissão de vídeo através de chaves ATM.

Em [OLI 95] foi descrito um modelo de tarefas imprecisas para ambiente distribuído. Embora o artigo discuta a problemática do escalonamento neste contexto, algoritmos de escalonamento propriamente ditos não são fornecidos para o modelo de tarefas apresentado.

2.5 Conclusões

Neste capítulo foi feita uma revisão de sistemas tempo real e da problemática do escalonamento tempo real. A seção 2.3 procurou sintetizar a discussão existente no que diz respeito às abordagens para este problema e suas caracterísitcas. Uma das abordagens possíveis, a Computação Imprecisa, foi discutida com maiores detalhes na seção 2.4.

Com respeito às aplicações tempo real distribuídas, especialmente aplicações encontradas na automação industrial, o mais razoável é esperar conjuntos mistos de tarefas. Aplicações industriais nem sempre apresentam tarefas críticas com respeito à segurança ("safety-critical"). Mas quase sempre existirão tarefas críticas à missão. O usuário da aplicação certamente gostaria de garantias, em tempo de projeto, para as tarefas críticas à missão, mesmo que esta garantia signifique um gasto adicional de recursos. Entretanto, para

a maioria das tarefas do sistema, provavelmente uma abordagem tipo melhor esforço ("best-effort") seria suficiente.

Existe um dilema na área de escalonamento tempo real, entre ter garantia para o prazo das tarefas ou ter adaptabilidade e eficiência no uso dos recursos. Se o aspecto flexibilidade na carga for ignorado, sempre é possível tratar todas as tarefas no sentido "garantia". Mas esta abordagem muitas vezes é inviável, tal o volume de recursos necessários para o pior caso de todas as tarefas. Ela é viável apenas em sistemas muito pequenos ou em sistemas críticos ao ponto de justificar todos os recursos gastos. Por outro lado, se a necessidade de garantia for ignorada, é possível tratar todas as tarefas no sentido "melhor esforço". Mas isto significa não ter qualquer garantia temporal para a aplicação. Estas considerações são válidas tanto no contexto centralizado como no distribuído. No ambiente distribuído, os algoritmos ainda tendem a ser mais complexos e computacionalmente mais custosos.

A técnica Computação Imprecisa parece oferecer uma das possíveis soluções de compromisso para este dilema. Os trabalhos sobre Computação Imprecisa já publicados tratam, em geral, apenas do problema de escalonamento local. Mesmo assim, somente empregando modelos de tarefas relativamente simples. Diversos aspectos que são inerentes ao contexto distribuído nunca foram abordados na literatura relativa à Computação Imprecisa. Em resumo, não existem trabalhos publicados que mostram como aplicações de tempo real, construídas a partir do conceito de Computação Imprecisa, podem ser escalonadas em ambiente distribuído.

3 Escalonamento de Tarefas Imprecisas em Ambiente Distribuído

3.1 Introdução

O objetivo geral desta tese é mostrar como aplicações de tempo real, construídas a partir do conceito de Computação Imprecisa, podem ser escalonadas em ambiente distribuído. Como foi dito no capítulo anterior, poucos trabalhos existentes na literatura tratam de Computação Imprecisa em ambiente distribuído. Em particular, diversos aspectos que são inerentes a este contexto nunca foram abordados na literatura.

Este problema geral na verdade engloba diversos problemas específicos, razoavelmente independentes entre si. O objetivo deste capítulo é mostrar os diversos aspectos envolvidos e descrever a abordagem geral adotada. Cada um dos quatro capítulos seguintes da tese discutirá um aspecto em particular, descrevendo uma abordagem específica para o aspecto em questão e apresentando uma solução de escalonamento propriamente dita.

Neste capítulo é inicialmente discutida a problemática do escalonamento tempo real de tarefas imprecisas em ambiente distribuído. Em seguida, é descrita a abordagem adotada neste trabalho para lidar com os problema apontados. A adoção de uma abordagem permite então a descrição do modelo de tarefas básico apropriado. É para este modelo de tarefas que as soluções de escalonamento serão construidas, nos capítulos seguintes da tese.

3.2 Discussão da Problemática

A problemática do escalonamento tempo real de tarefas imprecisas em ambiente distribuído envolve diversos aspectos. A discussão está dividida em quatro aspectos, onde são identificados os problemas principais.

Alocação

No contexto de sistemas distribuídos, o escalonamento tempo real é geralmente resolvido em duas etapas. Na primeira etapa é feita a alocação das tarefas aos processadores. Na segunda etapa é feito o escalonamento local de cada processador, tomado individualmente. Este escalonamento local considera como carga as tarefas alocadas na primeira etapa.

Em aplicações de tempo real geralmente não existe migração de tarefas em tempo de execução, pois esta técnica demanda recursos tanto de processamento como de comunicação e torna a tarefa sendo migrada temporariamente indisponível. Como cada tarefa é alocada permanentemente a um processador, aumenta a importância de uma alocação adequada.

Muitas vezes uma tarefa já está associada com um processador em particular como requisito de projeto. Por exemplo, tarefas que precisam acessar recurso computacional disponível em apenas alguns dos processadores, como um coprocessador gráfico. Neste

caso, a tarefa deve ser necessariamente alocada a um dos processadores que dispõe do coprocessador em questão. Outro caso são tarefas que atuam diretamente sobre o ambiente através de algum dispositivo físico (válvula, relé, etc). Neste caso, a tarefa deve ser alocada necessariamente ao processador onde se encontra o dispositivo físico.

Também é comum especificar no projeto que determinado subconjunto de tarefas deve ficar necessariamente no mesmo processador. Isto decorre da forte interação entre elas e da conseqüente vantagem em mantê-las no mesmo processador. Isto permite, entre outras coisas, que a comunicação entre elas seja feita através de memória compartilhada. Técnicas para comunicação entre tarefas baseadas em memória compartilhada são mais eficientes que o envio de mensagens, mas exigem que as tarefas envolvidas residam no mesmo processador.

Algoritmos de tolerância a faltas normalmente empregam réplicas que devem ser executadas em processadores diferentes, com o objetivo de obter uma independência entre faltas. Neste caso, o algoritmo de alocação deve ser capaz de garantir que tarefas pertencentes a um mesmo grupo de réplicas sejam alocadas em processadores diferentes.

O objetivo primário da alocação, quando do uso de técnicas de Computação Imprecisa, é garantir que todas as tarefas sempre poderão concluir suas respectivas partes obrigatórias antes do deadline. Para isto o algoritmo de alocação deve tentar diversas soluções de alocação, verificando para cada uma delas a escalonabilidade das partes obrigatórias. Este problema possui complexidade exponencial em relação ao número de tarefas. Com respeito ao objetivo primário, qualquer solução de alocação que permita uma garantia em projeto para as partes obrigatórias é igualmente satisfatória.

A alocação possui como objetivo secundário aumentar as chances das partes opcionais das tarefas serem executadas. Considere duas soluções de alocação X e Y, igualmente satisfatórias do ponto de vista do objetivo primário, isto é, partes obrigatórias podem ser garantidas tanto em X como em Y. É possível que a alocação X seja melhor balanceada com respeito a alocação Y, de forma que partes opcionais possam ser mais executadas em X do que em Y. Logo, do ponto de vista do objetivo secundário, a alocação X será superior à alocação Y. É importante notar que partes opcionais podem possuir diferentes graus de importância para o sistema. A comparação entre duas diferentes alocações do ponto de vista do objetivo secundário, quando possível, pode ser ponderada com respeito a importância das partes opcionais.

Garantia para as Partes Obrigatórias

Dentro do contexto da Computação Imprecisa, é necessário ainda em tempo de projeto garantir que cada parte obrigatória será concluída dentro do seu respectivo deadline. Ao tratar deste aspecto, é possível ignorar a existência das partes opcionais e considerar cada tarefa formada exclusivamente por sua parte obrigatória.

Existem na literatura inúmeras soluções de escalonamento para oferecer garantia quando um conjunto de tarefas executa em um único processador. Para sistemas distribuídos os trabalhos existentes são em muito menor número, em função dos problemas adicionais encontrados. Uma previsibilidade determinista para as partes obrigatórias pode ser obtida, entre outras formas, através da abordagem "teste de escalonabilidade + prioridades" (seção 2.3). Neste tipo de escalonamento, baseado em prioridades ([LIU 73]),

tarefas recebem prioridades segundo alguma política específica e um teste de escalonabilidade é executado em tempo de projeto, determinando se existe a garantia de que todas as tarefas serão executadas dentro dos deadlines. O teste de escalonabilidade deve ser compatível com a política de atribuição de prioridades adotada. Em tempo de execução, um escalonador preemptivo produz a escala de execução usando as prioridades das tarefas.

Uma necessidade básica neste contexto são protocolos de comunicação com atraso limitado. Protocolos de comunicação tempo real fogem do escopo deste trabalho e não serão abordados. Será suposta a existência de um protocolo capaz de garantir que o envio de uma mensagem entre dois processadores distintos demora, no máximo, Δ unidades de tempo.

Em sistemas distribuídos também é comum o surgimento de relações de precedência entre tarefas. Uma relação de precedência tem origem em uma necessidade de sincronização e/ou passagem de dados entre duas tarefas da aplicação. Uma mensagem cria uma relação de dependência entre a tarefa remetente e a tarefa destinatária. A tarefa destinatária somente pode iniciar sua execução após receber a mensagem. Embora isto ocorra também em sistemas centralizados, o fato das tarefas estarem distribuídas por diferentes processadores dificulta o escalonamento no sistema. Qualquer análise de escalonabilidade para ambiente distribuído deve ser capaz de lidar com relações de precedência.

É possível empregar deslocamentos temporais ("offsets", [AUD 93b]) para implementar relações de precedência entre tarefas. Ao estabelecer um deslocamento temporal ("offset") entre as liberações de duas tarefas, é possível garantir que a tarefa sucessora somente iniciará sua execução após o término da tarefa predecessora. Basta que este deslocamento temporal seja maior ou igual ao tempo máximo de resposta da tarefa predecessora. No caso de haver passagem de dados, no instante que a tarefa sucessora é liberada é garantido que os dados já foram colocados no lugar apropriado pela tarefa predecessora. Esta técnica é algumas vezes chamada de liberação estática de tarefas ("statically released tasks", [SUN 95]), pois o instante de liberação das tarefas é previamente fixado. O sistema original é transformado em um sistema equivalente onde as tarefas são independentes mas apresentam um deslocamento entre si. Testes de escalonabilidade são então aplicados a este sistema equivalente.

Também é possível implementar relações de precedência através do envio explícito de uma mensagem da tarefa predecessora para a tarefa sucessora. Esta mensagem informa que a tarefa predecessora foi concluída e a tarefa sucessora pode ser liberada. A mensagem pode também conter os dados a serem transmitidos de uma tarefa para a outra. Esta técnica é algumas vezes chamada de liberação dinâmica de tarefas ("dynamically released tasks", [SUN 95]), pois o instante de liberação da tarefa sucessora depende do instante de conclusão da tarefa predecessora, o qual somente é conhecido em tempo de execução. A incerteza a respeito do instante de liberação da tarefa sucessora pode ser modelada como um *jitter* de liberação ("release jitter", [TIN 94c]). Para fins de análise, o sistema original é transformado em um sistema equivalente onde as tarefas são independentes mas apresentam uma variação no instante de liberação. Novamente, testes de escalonabilidade são aplicados ao sistema equivalente.

Relações de precedência simples, envolvendo apenas duas tarefas, podem ser adequadamente modeladas através da simples introdução de *jitter* na liberação. A tarefa destinatária sofre um atraso em sua liberação devido à espera pela mensagem. Entretanto,

padrões mais complexos de precedência, na forma de grafos acíclicos dirigidos, exigem um tratamento mais elaborado. Como será mostrado no capítulo 4, o simples emprego de *jitter* na liberação resulta em uma análise muito mais pessimista que o verdadeiro pior caso.

Identificação de Folgas na Escala de Execução

Garantir, em tempo de projeto, que todas as partes obrigatórias serão concluidas antes do respectivo deadline implica em reservar recursos para o pior caso. Pior caso aqui se refere tanto aos tempos de execução quanto a pior combinação possível de liberações de tarefas. Em tempo de execução, existem vários fatores capazes de gerar folgas na utilização dos recursos. Estas folgas podem ser usadas para executar as partes opcionais.

O tempo de processador que sobra após a reserva de recursos, para o pior caso de todas as partes obrigatórias, é chamado de capacidade restante. Esta capacidade restante é normalmente medida em termos de utilização do processador. Ela pode ser dedicada à execução de partes opcionais sem prejuizo para as partes obrigatórias. Sua grandeza, em termos de utilização do processador, já é conhecida em tempo de projeto.

A probabilidade do pior caso acontecer é, em geral, muito pequena. Na maior parte das vezes, as partes obrigatórias podem ser concluidas antes do deadline com menos recursos do que foi reservado em tempo projeto. Isto acontece quando uma determinada ativação de tarefa ocupa menos tempo de processador do que o pior caso previsto em projeto. O tempo de processador que foi reservado em tempo de projeto mas não utilizado em tempo de execução é chamado de tempo ganho ("gain time"). Sua grandeza e instante de ocorrência somente é conhecida em tempo de execução, quando o verdadeiro tempo de processador necessário para cada ativação de cada tarefa é identificado.

É também possível que uma dada tarefa seja ativada com uma frequência menor do que a frequência máxima estabelecida em projeto. Desta forma, a parte obrigatória desta tarefa vai consumir menos recursos do que no pior caso. Este tempo de processador reservado mas não utilizado em função de uma menor frequência de ativação também pode ser usado na execução de partes opcionais. Sua grandeza e instante de ocorrência somente é conhecida em tempo de execução, quando o instante exato das chegadas de cada tarefa é identificado.

Os testes de escalonabilidade aplicados em tempo de projeto sempre consideram, na sua análise, a pior combinação possível para a liberação das tarefas. Por exemplo, os testes baseados no conceito do instante crítico reservam recursos para quando todas as tarefas são liberadas simultaneamente. Sempre que as tarefas não forem liberadas simultaneamente, haverá uma sobra de recursos em tempo de execução. Isto acontece com frequência devido as tarefas esporádicas, as tarefas periódicas com períodos diferentes entre si, ao *jitter* de liberação e as relações de precedência. Nestas situações, recursos estarão sendo liberados para a execução de partes opcionais.

O somatório de todas estas fontes de recurso resulta na capacidade ociosa do sistema ("spare capacity"). Esta capacidade ociosa deve ser detectada e usada para a execução das partes opcionais. É necessário que as partes opcionais executadas ocupem somente a capacidade ociosa, para não comprometer a execução das partes obrigatórias previamente garantidas.

A cada instante t, a capacidade ociosa do sistema dá origem ao tempo de folga S(t) ("slack time"). A quantidade S(t) representa quanto do tempo do processador pode ser ocupado no instante t sem comprometer as partes obrigatórias garantidas antes. O problema da identificação de folgas na escala de execução se resume então a identificação de S(t) no sistema em questão.

Como será visto no capítulo 5, algoritmos para a detecção e o aproveitamento da capacidade ociosa tendem a ser simples e pouco eficientes ou então eficientes porém complexos. Este problema é agravado pelo fato de muitos destes algoritmos serem empregados em tempo de execução, quando eles disputam recursos (processador) com as próprias partes opcionais.

Escolha das Partes Opcionais para Execução

Muitas vezes a folga identificada na escala de execução não é suficiente para executar todas as partes opcionais do sistema. Desta forma, é necessário escolher quais das partes opcionais disponíveis para execução em um dado momento deverão ser efetivamente executadas e quais deverão ser descartadas. Esta escolha deve levar em consideração a importância de cada tarefa para o sistema e a quantidade de recurso (tempo de processador) necessário.

Quando múltiplas versões são usadas, é necessário decidir qual versão de uma tarefa executará antes da tarefa começar. Uma vez que a tarefa iniciou sua execução não é mais possível rever aquela decisão. Temos também que a tarefa apresenta uma restrição 0/1, ou seja, a parte opcional será completamente executada (quando a versão precisa é escolhida) ou não será executada (quando a versão imprecisa é escolhida).

Em muitos sistemas a importância de executar precisamente uma determinada tarefa depende do comportamento passado do sistema. Uma situação comum são tarefas periódicas onde a importância de uma execução precisa aumenta na medida que a tarefa é executada imprecisamente. Neste tipo de tarefa a imprecisão é, de certa maneira, cumulativa. É desejável que a tarefa seja algumas vezes executada de forma precisa para interromper o acumulo de imprecisão. Este tipo de dependência é chamada de intra-tarefa ([OLI 96b]). Exemplos de tarefas com dependências intra-tarefa aparecem em aplicações tais como sistemas de radar e sistemas de controle ([CHU 90]).

Tarefas que apresentam uma relação de produtor e consumidor também apresentam uma dependência com respeito a importância de uma execução precisa. A tarefa produtora gera como saída dados que serão usados como entrada pela tarefa consumidora. Muitos algoritmos apresentam um menor tempo de execução quando os dados recebidos são de melhor qualidade. Desta forma, ao executar precisamente a tarefa produtora estaremos diminuindo o tempo de execução da tarefa consumidora. Este "tempo extra de processador" gerado poderá ser utilizado então para executar outras partes opcionais, aumentando a utilidade do sistema. Este tipo de dependência é chamada de inter-tarefa ([OLI 96b]). Exemplos de tarefas com dependências inter-tarefa aparecem em aplicações tais como reconhecimento de voz, sistemas de radar e processamento de imagem para navegação autônoma ([FEN 94], [KRO 94]).

A maioria dos estudos de computação imprecisa considera que o valor/erro de uma tarefa é definido no seu intante de chegada. É suposto que ela não é afetada pelo que

acontece a outras tarefas ou a liberações prévias da mesma tarefa. Os trabalhos [FEN 94] e [CHU 90] constituem uma exceção. Em [FEN 94] o modelo de tarefas inclui dependências inter-tarefa. Erros nos dados de entrada de uma tarefa podem estender o tempo máximo de execução de suas próprias partes obrigatórias e/ou opcionais. Em [CHU 90] o modelo de tarefas inclui dependências intra-tarefa; é suposto que todas as tarefas devem ser executadas precisamente pelo menos uma vez a cada **Q** liberações. Nos dois trabalhos citados o modelo de erro é tal que uma parte opcional poderia eventualmente tornar-se obrigatória ou aumentar o tempo global de execução obrigatória do sistema, perdendo assim seu carater "opcional".

O modelo de tarefas a ser adotado deve ser tal que as tarefas imprecisas possuam dependências intra-tarefa e inter-tarefa. Diferentemente do trabalho em [FEN 94] e [CHU 90], as partes opcionais no modelo usado devem permanecer sempre opcionais. Mesmo se nunca executadas, elas não aumentam o tempo de execução obrigatória de qualquer tarefa do sistema nem tornam-se, em qualquer sentido, obrigatórias. Não é do conhecimento do autor outros trabalhos onde tarefas imprecisas, com o tipo de dependência apresentado aqui, tenham sido estudadas.

Algoritmos para escolha de partes opcionais devem contemplar todos estes aspectos descritos acima. Em particular, as dependências intra-tarefa e inter-tarefa devem ser consideradas no momento de comparar a importância para o sistema da execução precisa de diferentes tarefas. Como tais algoritmos são executados on-line, é importante que seu custo computacional seja baixo. Caso contrário, qualquer ganho devido ao uso de um algoritmo melhor será perdido em função do tempo de processador gasto para executar o próprio algoritmo.

3.3 Abordagem adotada

Nesta seção será descrita a abordagem adotada nesta tese para escalonar tarefas imprecisas em ambiente distribuído. O escalonamento neste trabalho é definido com base em abordagens presentes na literatura ([AUD 94a], [RAM 90]) que tratam com cargas mistas. Isto é, aplicações que possuem uma mistura de tarefas críticas com tarefas não críticas. Tais abordagens foram adaptadas com a inclusão da Computação Imprecisa. Não é possível comparar a abordagem adotada com propostas alternativas uma vez que inexistem na literatura outras abordagens completas para o escalonamento de tarefas imprecisas em ambiente distribuído. Entretanto, os algoritmos propostos para a abordagem adotada serão, estes sim, avaliados ou comparados com soluções alternativas.

Nos últimos anos, resultados teóricos importantes foram obtidos a partir de algoritmos de escalonamento que trabalham com prioridades, especialmente prioridades fixas. O emprego desta técnica de escalonamento permite garantir, em tempo de projeto, que todas as partes obrigatórias serão executadas antes do respectivo deadline. A manipulação de prioridades também oferece uma forma simples de tratar das partes opcionais. Em função da literatura existente, a abordagem adotada faz uso do escalonamento por prioridades fixas.

Este estudo se limita à técnica de múltiplas versões onde duas versões são usadas na programação de tarefas imprecisas. Múltiplas versões é a forma mais flexível para a programação de tarefas imprecisas. Por outro lado, funções monotônicas e funções

melhoramento também podem ser consideradas formas especiais de múltiplas versões. No caso de função monotônica, a função completa representa a versão primária enquanto que a função com tempo mínimo de execução representa a versão secundária. Uma função melhoramento pode ser vista como uma tarefa imprecisa com parte obrigatória nula. Desta forma, deste ponto em diante será suposto que todas as tarefas imprecisas foram programadas com múltiplas versões e apresentam restrição 0/1.

A figura 3.1 ilustra a abordagem adotada, composta por quatro algoritmos. O algoritmo 1 realiza a alocação das tarefas aos processadores; o algoritmo 2 é capaz de verificar se as partes obrigatórias serão sempre concluídas antes do respectivo deadline; o algoritmo 3 realiza a escolha de quais partes opcionais, em um dado momento, devem ser consideradas para execução; finalmente, o algoritmo 4 é capaz de garantir que uma dada parte opcional pode ser executada sem prejuízo para as partes obrigatórias e opcionais previamente garantidas.

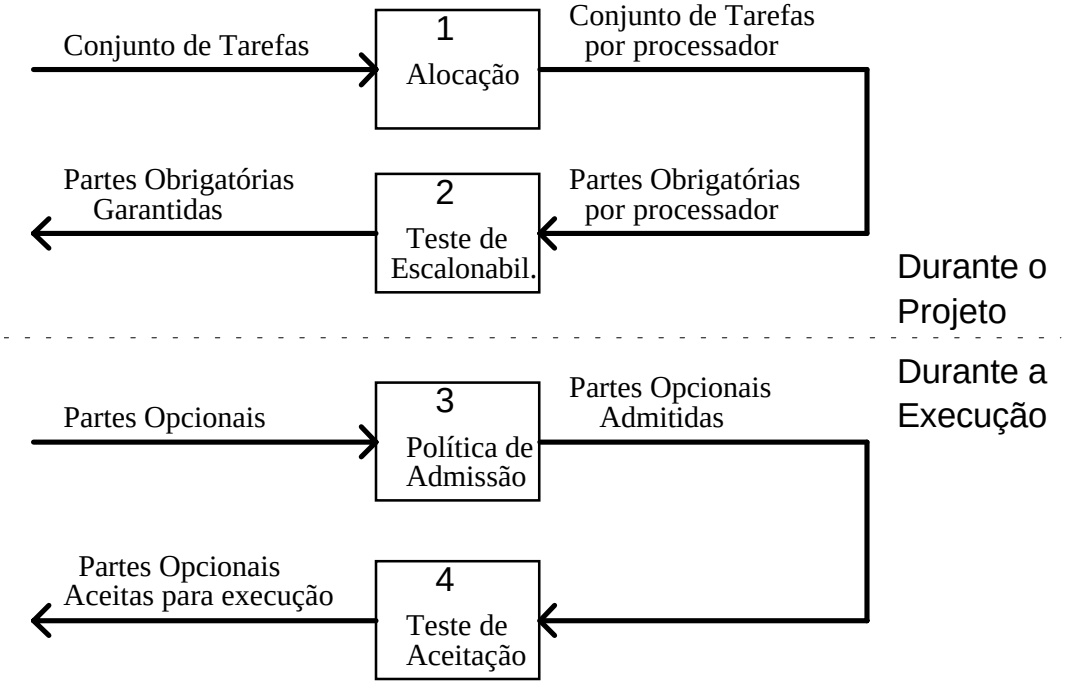

Figura 3.1 - Abordagem geral adotada.

O primeiro algoritmo corresponde ao processo de alocação das tarefas aos processadores. O conjunto inicial de tarefas é particionado em **h** subconjuntos, onde **h** representa o número de processadores do sistema. O algoritmo empregado nesta etapa deve ser capaz de lidar com restrições do tipo "tarefas X e Y devem ficar no mesmo processador", "tarefas X e Y não podem ficar no mesmo processador" ou ainda "tarefa X deve ficar nos processadores A, B ou C". Uma alocação de sucesso exige que todas as partes obrigatórias sejam garantidas. Logo, o algoritmo de alocação deve usar o algoritmo 2 para verificar se este objetivo foi atendido. Ao mesmo tempo, o algoritmo de alocação procura obter um balanceamento de carga de tal sorte que as chances de uma parte opcional executar sejam ampliadas.

O segundo algoritmo deve ser capaz de verificar se todos os deadlines serão cumpridos quando somente as partes obrigatórias são executadas. Uma vez que prioridades fixas serão empregadas, é necessário determinar uma política para a atribuição de prioridades e então um teste de escalonabilidade apropriado. O problema é complicado pelo

fato de tarefas em diferentes processadores se comunicarem através de mensagens. Logo, é possível que a escala de execução de um processador **A** seja afetada pelos atrasos decorrentes de mensagens que são enviadas de um outro processador **B** para uma de suas tarefas.

O terceiro algoritmo avalia, em tempo de execução, se uma dada parte opcional deve ser ou não considerada para execução. A idéia aqui é implementar uma política de admissão que descarta partes opcionais cuja importância para o sistema é muito baixa. Obviamente, o nível mínimo de importância para uma parte opcional ser admitida depende da carga do processador a cada instante.

O quarto algoritmo avalia se uma dada parte opcional, previamente admitida pelo algoritmo 3, pode ser aceita para execução. Uma parte opcional somente poderá ser aceita para execução se não comprometer as garantias fornecidas anteriormente. O algoritmo 4 deve levar em conta as condições supostas pelo algoritmo 2. Em outras palavras, uma parte opcional pode ser aceita se ela não alterar as condições supostas pelo algoritmo 2 de tal forma a invalidar o teste de escalonabilidade feito em tempo de projeto.

O objetivo geral desta tese é mostrar como aplicações de tempo real, construídas a partir do conceito de Computação Imprecisa, podem ser escalonadas em ambiente distribuído. Uma vez definida a abordagem que será empregada, resta resolver os quatro problemas de escalonamento. Isto é:

- ALOCAÇÃO: Como resolver qual tarefa executa em qual processador, de forma que todas as partes obrigatórias das tarefas possam ser garantidas e que as partes opcionais estejam distribuídas de forma que sua chance de execução seja maximizada.

- TESTE DE ESCALONABILIDADE: Como garantir que as partes obrigatórias das tarefas serão concluídas antes dos respectivos deadlines, em um ambiente onde tarefas podem executar em diferentes processadores e o emprego de mensagens cria relações de precedência entre elas.

- POLÍTICA DE ADMISSÃO: Como escolher quais partes opcionais devem ser executadas, na medida em que o recurso "tempo de processador disponível" não permite a execução de todas as partes opcionais da aplicação.

- TESTE DE ACEITAÇÃO: Como determinar que a execução de uma parte opcional não irá comprometer a execução das partes obrigatórias, previamente garantidas, violando as premissas usadas para fornecer tal garantia.

Ao resolver estes quatro problemas de escalonamento estará sendo mostrado que efetivamente Computação Imprecisa pode ser usada como base para a construção de aplicações distribuídas de tempo real.

3.4 Modelo Básico de Tarefas Adotado

Esta seção descreve o modelo básico de tarefas que foi adotado e será usado nos capítulos seguintes. São descritas as propriedades usuais associadas com tarefas em sistemas de tempo real. Para efeito de clareza do texto, alguns detalhes do modelo de

tarefas serão somente apresentados nos capítulos que tratam especificamente dos algoritmos de escalonamento afetados.

O modelo de tarefas empregado supõe a execução da aplicação em um sistema distribuído formado por um conjunto **H** composto por **h** processadores (ou nós), conectados através de um meio de comunicação. O envio de mensagens entre dois processadores quaisquer (comunicação remota) é delimitado por um tempo máximo **Δ**. É suposto que o protocolo de comunicação será executado por um processador auxiliar, não competindo portanto com as tarefas da aplicação. O tempo para o envio de uma mensagem entre duas tarefas no mesmo processador será considerado zero para efeito de escalonamento.

Uma aplicação distribuída neste modelo é expressa na forma de um conjunto de atividades. A atividade é a entidade encapsuladora de tarefas que se comunicam e/ou se sincronizam. Cada atividade pode ser representada por um grafo orientado acíclico, correspondendo a um conjunto de tarefas. Os nodos desde grafo representam tarefas e os arcos são relações de precedência. Uma relação de precedência direta tem origem em uma necessidade de sincronização e/ou passagem de dados entre duas tarefas de uma mesma atividade. Uma relação de precedência direta não será satisfeita de forma implícita, através da passagem de tempo. Ela implica no envio de uma mensagem explícita da tarefa predecessora para a tarefa sucessora.

Uma aplicação é formada por um conjunto **A** de atividades contendo **m** atividades que podem ser periódicas ou esporádicas. Cada atividade A_X, $1 \leq x \leq m$, pertencente ao conjunto **A**, está sujeita a uma sequência infinita de ativações. Cada atividade também está associada a um conjunto de tarefas possivelmente ligadas entre si por relações de precedência. Com um ligeiro abuso da notação, vamos também chamar de A_X o conjunto das tarefas que formam a atividade A_X.

Uma atividade periódica A_X qualquer é caracterizada por um período P_X, ou seja, um intervalo regular de tempo entre ativações sucessivas. É suposto que a primeira ativação de todas as atividades periódicas ocorre no instante zero de tempo. Atividades periódicas podem possuir várias tarefas iniciais, ou seja, várias tarefas sem predecessores.

Uma atividade esporádica A_X qualquer é caracterizada por um intervalo mínimo de tempo entre ativações P_X. Também é suposto que atividades esporádicas possuem uma única tarefa inicial, ou seja, possuem uma única tarefa sem predecessores.

Toda aplicação está associada a um conjunto **T** contendo **n** tarefas, ou seja, o conjunto contendo todas as tarefas pertencentes a todas as atividades do conjunto **A**. As tarefas não possuem relações de exclusão mútua entre si. Qualquer tarefa pode ser preemptada, ou seja, ter sua execução suspensa e retomada mais tarde. Cada tarefa T_i é caracterizada por um *jitter* de liberação máximo J_i, uma prioridade global $\rho(T_i)$ e um deadline D_i, relativo ao início da sua atividade. Para toda tarefa T_i, $1 \leq i \leq n$, $T_i \in Ax$, temos que $D_i \leq P_X$.

As tarefas recebem individualmente uma prioridade única na aplicação. Sem perda de generalidade vamos supor que as tarefas foram numeradas na ordem decrescente de suas

prioridades, de maneira que se $i<j$ então $\rho(T_i)>\rho(T_j)$, ou seja, T_i possui prioridade maior que T_j.

Cada tarefa T_i é composta por uma parte obrigatória e uma parte opcional. O tempo de execução no pior caso de ambas as partes é conhecido em tempo de projeto e denotado por M_i e O_i, respectivamente. É também suposto que existe uma restrição 0/1 para as partes opcionais. Isto é, cada parte opcional deve ser executada completamente ou não iniciada. Esta decisão deve ser tomada antes de iniciar a execução da tarefa.

Cada tarefa T_i é também caracterizada pelo conjunto $dPred(T_i)$, que contém todas as tarefas que são predecessoras diretas de T_i. A tarefa T_i não inicia sua execução até receber uma mensagem de cada uma de suas predecessoras diretas. A partir do conjunto $dPred(T_i)$ de cada tarefa T_i, vamos definir mais cinco tipos de conjuntos:

- Conjunto $iPred(T_i)$ das tarefas predecessoras indiretas de T_i:

$\forall\ T_j \in T,$

$T_j \in iPred(T_i) \Leftrightarrow T_j \notin dPred(T_i) \wedge \exists T_k \in T . T_j \in dPred(T_k) \wedge T_k \in dPred(T_i) \cup iPred(T_i).$

- Conjunto $Pred(T_i)$ das tarefas predecessoras diretas ou indiretas de T_i:

$\forall\ T_j \in T,$

$T_j \in Pred(T_i) \Leftrightarrow T_j \in dPred(T_i) \cup iPred(T_i).$

- Conjunto $dSuc(T_i)$ das tarefas sucessoras diretas de T_i:

$\forall\ T_j \in T,\ T_j \in dSuc(T_i) \Leftrightarrow T_i \in dPred(T_j).$

- Conjunto $iSuc(T_i)$ das tarefas sucessoras indiretas de T_i:

$\forall\ T_j \in T,$

$T_j \in iSuc(T_i) \Leftrightarrow T_j \notin dSuc(T_i)\ \wedge \exists T_k \in T . T_j \in dSuc(T_k) \wedge T_k \in dSuc(T_i) \cup iSuc(T_i).$

- Conjunto $Suc(T_i)$ das tarefas sucessoras diretas ou indiretas de T_i:

$\forall\ T_j \in T,$

$T_j \in Suc(T_i) \Leftrightarrow T_j \in dSuc(T_i) \cup iSuc(T_i).$

Quando duas tarefas T_i e T_j não guardam qualquer relação de precedência, direta ou indireta, elas são ditas independentes entre si. Este fato é denotado por $T_i // T_j$. Em outras palavras, $\forall\ T_i \in T, \forall\ T_j \in T,\ T_i // T_j \Leftrightarrow T_i \notin Pred(T_j) \cup Suc(T_j)$.

Pela definição de atividade, para $\forall\ T_i \in T$, temos que:
$\exists T_j \in A_X\ .\ T_i \in Pred(T_j) \cup Suc(T_j) \Rightarrow T_i \in A_X$. Observe que a recíproca não é verdadeira, pois duas tarefas independentes entre si podem fazer parte de uma mesma atividade.

Não existe migração de tarefas em tempo de execução. Nada impede que, como resultado da alocação, tarefas de uma mesma atividade fiquem em processadores diferentes. Atividades que se estendem por mais de um processador encapsulam comunicações remotas, determinando então a extensão de grafos de precedência por mais de um escalonador local.

Caso a tarefa T_i seja uma tarefa inicial de sua atividade, ela poderá possuir um *jitter* de liberação intrínsico J_i maior que zero. É suposto que, quando duas tarefas T_i, T_k são tarefas iniciais de uma mesma atividade A_X, então $J_i = J_k$. Este *jitter* de liberação intrínsico

pode surgir quando uma atividade é ativada pela ocorrência de um evento externo, o qual é amostrado de tempos em tempos pelo sistema. Passa a existir um atraso entre a ocorrência do evento externo e a liberação da respectiva atividade, que somente ocorre após a amostragem do evento. O *jitter* de liberação também pode surgir devido ao problema conhecido como "tick scheduling" ([TIN 94b]) no caso de escalonadores ativados pela passagem do tempo. Neste caso, o exato instante de tempo no qual a atividade deveria ser liberada não coincide com uma interrupção do relógio do sistema. Desta forma, ocorre um atraso na sua liberação. Tarefas que possuem predecessores são liberadas em função da recepção de uma mensagem e não apresentam *jitter* de liberação intrínsico mas apenas o *jitter* de liberação decorrente do atraso variável na recepção das mensagens.

Como em [DAV 95], ao longo deste trabalho vamos empregar V(t) para denotar o valor total do sistema até o instante **t**, dado pelo somatório dos valores adicionados por todas as liberações de todas as tarefas. O valor $\Phi(t)$ representa o tempo total de processador, dentro do intervalo [0,t), que não foi usado para executar partes obrigatórias. A densidade média de valor obtida pelo sistema até o instante **t** será denotada por $\Lambda(t)$ e calculada da seguinte forma: $\Lambda(t) = V(t) / \Phi(t)$.

É suposto que cada tarefa T_i está associada com um valor nominal V_i. O valor nominal V_i representa a utilidade da tarefa T_i no sistema. Entretanto, o valor nominal não considera as dependências entre tarefas. Esta tese introduz o conceito de valor efetivo. Em tempo de execução, a cada liberação $T_{i,k}$ da tarefa T_i é associado um valor efetivo $V_{i,k}$. O valor efetivo é calculado a partir do valor nominal e considera as dependências entre tarefas. O valor adicionado por $T_{i,k}$ ao sistema será zero no caso de uma execução imprecisa ou $V_{i,k}$ quando a parte opcional for executada. O objetivo do escalonamento das partes opcionais é maximizar o valor total do sistema.

A dependência intra-tarefa está associada com a importância de ocasionalmente executar uma dada tarefa precisamente. Sendo assim, a dependência intra-tarefa entre as liberações $T_{i,k}$ e $T_{i,k+1}$, de uma dada tarefa T_i, é modelada assumindo-se que uma execução imprecisa de $T_{i,k}$ vai aumentar o valor efetivo de uma execução precisa da liberação $T_{i,k+1}$. O valor efetivo da liberação $T_{i,k+1}$, denotado por $V_{i,k+1}$, será:

V_i	quando a execução de $T_{i,k}$ é precisa;
$V_i + (\alpha_i . V_{i,k})$	quando a execução de $T_{i,k}$ é imprecisa;

onde α_i é chamada de taxa de recuperação da tarefa T_i. Ela descreve a quantidade de valor que pode ser recuperada após uma dada liberação não ter sido executada precisamente. O fato de uma tarefa T_j qualquer não apresentar dependência intra-tarefa é facilmente modelado por $\alpha_j=0$. Embora o conceito de dependência intra-tarefa já estivesse implícito em [CHU 90], a formulação apresentada neste trabalho, baseada em taxa de recuperação, é original.

A dependência inter-tarefa está associada ao fato de dados de entrada com melhor qualidade reduzirem o tempo de execução de algumas tarefas. Estes dados de entrada são, muitas vezes, o produto da execução de uma tarefa predecessora direta. Desta forma, vamos supor que as dependências inter-tarefa surgem entre tarefas com relações de precedência direta. Uma tarefa predecessora é executada antes e poderá interferir na execução das tarefas que pertencem ao conjunto de suas sucessoras diretas.

A dependência inter-tarefa entre as liberações $T_{j,k}$ e $T_{i,k}$ é modelada assumindo-se que uma execução precisa de $T_{j,k}$ reduzirá o tempo de execução no pior caso das partes obrigatória e opcional de $T_{i,k}$. Em outras palavras, o tempo de execução de $T_{i,k}$ no pior caso será:

$M_i + O_i$	quando a execução de $T_{j,k}$ é imprecisa;
$\beta_{j,i} . M_i + \gamma_{j,i} . O_i$	quando a execução de $T_{j,k}$ é precisa;

onde $\beta_{j,i}$ e $\gamma_{j,i}$, $0<\beta_{j,i}\leq 1$, $0<\gamma_{j,i}\leq 1$, são os fatores de redução ligando T_j a T_i. Caso o tempo de execução de T_i dependa de mais de uma tarefa, é necessário aplicar os fatores de redução de todas as tarefas que interferem em T_i e executarem precisamente. Esta formulação para a dependência inter-tarefa é uma adaptação da formulação apresentada em [FEN 94].

Considerando a liberação $T_{i,k}$ da tarefa T_i, esta tese define a densidade de valor $\lambda_{i,k}$ de sua parte opcional como o seu valor efetivo dividido pelo seu tempo de execução no pior caso, após as correções em função das dependências inter-tarefa. Em outras palavras, temos:

$$\lambda_{i,k} = \frac{V_{i,k}}{O_i \times \prod_{\forall j.T_j \in [d\,Pred(T_i) \cap Opcio(T_{j,k})]} \gamma_{j,i}},$$

onde $Opcio(T_{j,k})$ representa o conjunto de tarefas cuja k-ésima liberação foi executada de forma precisa. O produtório no denominador reflete o fato de que várias tarefas podem preceder T_i e afetar seu comportamento. O cálculo de $\lambda_{i,k}$ considera o tempo máximo de execução da parte opcional de T_i após as correções devido às dependências inter-tarefa.

É importante observar que nenhum dos dois tipos de dependências definidos é capaz de aumentar a carga obrigatória do sistema. A dependência intra-tarefa resulta em aumento no valor efetivo de algumas partes opcionais. A dependência inter-tarefas resulta na redução do tempo máximo de execução de algumas tarefas. De qualquer modo, a carga obrigatória do sistema não aumenta. Esta propriedade é importante pois, caso contrário, o teste de escalonabilidade realizado em tempo de projeto perderia sua validade no momento em que a carga obrigatória do sistema aumentasse.

3.5 Conclusões

Neste capítulo foi feito inicialmente um levantamento dos problemas relacionados com o escalonamento tempo real de tarefas imprecisas em ambiente distribuído. A discussão da problemática foi dividida em quatro aspectos: alocação das tarefas aos processadores, garantia em tempo de projeto para as partes obrigatórias, identificação de folgas em tempo de execução e escolha de quais partes opcionais devem ser executadas quando não é possível executar todas.

Uma vez levantados os problemas, foi descrita a abordagem a ser adotada no sentido de atingir o objetivo geral da tese, isto é, escalonar tarefas imprecisas em ambiente

distribuído. Esta abordagem é composta por quatro algoritmos, associados com os quatro problemas específicos que surgem diretamente da discussão da problemática feita antes. São eles:

- Algoritmo de alocação;
- Teste de escalonabilidade;
- Política de admissão;
- Teste de aceitação.

No final do capítulo foi definido o modelo básico de tarefas a ser usado nos capítulos subsequentes, quando as soluções de escalonamento serão apresentadas. O modelo de tarefas inclui tarefas periódicas e esporádicas, ambiente distribuído, relações de precedência, *jitter* na liberação e deadline menor ou igual ao período. Tarefas possuem parte obrigatória e opcional, prioridades fixas e podem ser preteridas ("preemptive scheduler") a qualquer instante por uma tarefa de prioridade superior.

Além de apresentar as características temporais usuais, o modelo básico de tarefas adotado inova ao permitir que tarefas imprecisas possuam dependências tipo intra-tarefa e tipo inter-tarefa. Existe dependência intra-tarefa quando uma execução imprecisa aumenta a importância (o valor) de uma execução precisa da próxima ativação da mesma tarefa. Existe dependência inter-tarefa quando uma execução precisa da tarefa produtora dos dados diminui o tempo máximo de execução da tarefa consumidora destes mesmos dados. O modelo é tal que as dependências existentes não aumentam a carga obrigatória do sistema. As partes opcionais sempre mantém seu caracter opcional.

4 Teste de Escalonabilidade para Partes Obrigatórias

4.1 Introdução

A abordagem adotada neste trabalho para o escalonamento de tarefas imprecisas em ambiente distribuído requer, em tempo de projeto, a verificação da escalonabilidade da aplicação no que diz respeito as suas partes obrigatórias. Além das próprias restrições temporais de cada tarefa, a análise de escalonabilidade empregada deve considerar as relações de precedência criadas pelo envio de mensagens entre tarefas.

Neste momento da abordagem a preocupação é com a garantia da parte obrigatória de cada tarefa. As partes opcionais serão executadas com os recursos (tempo de processador) que sobrarem após esta garantia ter sido obtida. Desta forma, ao longo deste capítulo, as partes opcionais das tarefas serão ignoradas. Em particular, vamos usar C_i para denotar o tempo máximo de execução da tarefa T_i, sendo C_i definido por $C_i = M_i$.

O objetivo deste capítulo é apresentar uma solução para o emprego de prioridades fixas e liberação dinâmica de tarefas no escalonamento de sistemas tempo real críticos em ambiente distribuído. A partir de uma dada alocação, o método apresentado aqui verifica se o sistema é ou não escalonável. A seção 4.2 faz uma revisão da literatura relacionada com o assunto deste capítulo; a seção 4.3 detalha alguns aspectos da abordagem adotada; a seção 4.4 formaliza o problema; a seção 4.5 desenvolve regras de transformação que serão usadas na análise da escalonabilidade; a seção 4.6 sistematiza a aplicação das regras desenvolvidas e a seção 4.7 contém um exemplo que ilustra a vantagem da técnica desenvolvida.

4.2 Revisão da Literatura

A maior parte dos trabalhos publicados empregam prioridades fixas e liberação estática de tarefas para implementar relações de precedência. Relações de precedência arbitrárias em monoprocessadores são consideradas em [AUD 93b] e [GER 94]. O trabalho [BET 92] trata de relações de precedência arbitrárias em sistemas distribuídos.

Relações de precedência lineares ("pipelines"), onde cada tarefa possui no máximo um predecessor e um sucessor, são particularmente importantes em função de seu emprego em sistemas multimedia. Os trabalhos [SHA 94], [SUN 94], [CHA 95] e [SUN 95] tratam deste tipo de aplicação em ambiente multiprocessado. Todos empregam prioridades fixas e liberação estática de tarefas.

O trabalho [SUN 96b] apresenta algoritmos para o cálculo do tempo de resposta em sistemas distribuídos onde relações de precedência lineares são implementadas através da liberação dinâmica de tarefas. Em [SUN 96a] é feito um estudo comparativo entre o emprego de liberação dinâmica, de liberação estática e de uma abordagem mista para implementar relações de precedência lineares em ambiente distribuído.

Em [CHE 90a] é discutida a escalonabilidade de tarefas com relações de precedência em monoprocessador quando EDF é usado como algoritmo de escalonamento.

Em [HAR 94] é apresentada uma análise de escalonabilidade capaz de lidar com relações de precedência lineares em monoprocessador. Embora o modelo de tarefas usado inclua liberação dinâmica de tarefas, ou autores não usam *jitter* de liberação. Naquele trabalho é desenvolvido um conjunto de equações que descrevem especificamente o cenário estudado. Este conjunto de equações não utiliza o conceito *jitter* de liberação. O teste exato apresentado possui complexidade pseudo-polinomial. O modelo de tarefas usado não restringe a política de atribuição de prioridades e tarefas podem possuir deadlines arbitrários, ou seja, deadline menor, igual ou maior que o período da tarefa.

Em [AUD 93c] é sugerido o uso de liberação dinâmica de tarefas para implementar relações de precedência. É mostrado que *jitter* de liberação pode ser usado para modelar relações de precedência remota, ou seja, quando as tarefas relacionadas estão alocadas a diferentes processadores. Entretanto, o texto não desenvolve esta abordagem mais detalhadamente, argumentando que "a more detailed analysis of distributed precedence constrained tasks is beyond the scope of this paper".

Em [BUR 93] é considerada a análise de escalonabilidade em sistemas distribuídos com rede de comunicação ponto a ponto. Tarefas podem ser periódicas ou esporádicas e possuem prioridades fixas. Existem dois tipos de atividades, um tipo utiliza liberação dinâmica de tarefas ("loosely synchronous transaction") enquanto o outro tipo utiliza liberação estática de tarefas ("asynchronous transaction"). O modelo de tarefas adotado aceita relações arbitrárias de precedência e não restringe a política de atribuição de prioridades. Atividades como um todo podem ter deadline arbitrário, embora o intervalo de tempo entre a liberação de uma tarefa em particular e o término de sua execução deva ser menor ou igual ao seu período. O cálculo empregado neste artigo é tal que as relações de precedência que empregam liberação dinâmica de tarefa são transformadas, para todos os efeitos, em *jitter* de liberação e todas as tarefas passam a ser tratadas como tarefas independentes.

De forma semelhante, em [TIN 94c] é considerada a análise de escalonabilidade em sistemas distribuídos com rede de comunicação tipo barramento. São empregadas prioridades fixas e liberação dinâmica de tarefas. Relações de precedência são transformadas em *jitter* de liberação e todas as tarefas passam a ser tratadas como tarefas independentes. Tarefas podem ser periódicas ou esporádicas e possuem deadlines arbitrários. O modelo de tarefas adotado não restringe a política de atribuição de prioridades mas exige que cada tarefa receba no máximo uma mensagem, restringindo a análise às relações de precedência lineares. Relações de precedência arbitrárias são obtidas através da liberação estática de tarefas.

Neste trabalho serão utilizadas prioridades fixas e liberação dinâmica de tarefas para implementar relações arbitrárias de precedência. Vamos considerar deadline menor ou igual ao período para toda a atividade e usar *jitter* de liberação para modelar em parte o efeito das relações de precedência. Diferentemente de [BUR 93], vamos considerar também o efeito positivo das relações de precedência sobre o escalonamento, pois elas limitam os possíveis padrões de interferência. Desta forma, conjuntos de tarefas que seriam considerados inviáveis pela análise daquele trabalho, serão mostrados viáveis pela análise apresentada neste capítulo.

4.3 Descrição da Abordagem

Este capítulo trata do emprego de prioridades fixas e liberação dinâmica de tarefas para escalonar sistemas tempo real críticos em ambiente distribuído. A teoria de escalonamento relacionada com prioridades fixas evoluiu bastante nos últimos anos, passando a suportar modelos de tarefas bem mais complexos ([SHA 94], [AUD 93c]) que aqueles empregados nos primeiros trabalhos ([LIU 73]). Ao mesmo tempo, a liberação dinâmica de tarefas é mais flexível que a liberação estática. Por exemplo, a liberação dinâmica de tarefas facilita o escalonamento, em ambiente distribuído, de conjuntos de tarefas esporádicas com relações de precedência entre si. Além disto, o emprego de liberação estática de tarefas fragmenta as sobras de processador, tornando-as menos úteis para o escalonamento de partes opcionais. Estes fatores justificam a escolha da combinação de técnicas a ser empregada.

Neste trabalho vamos consider que tarefas recebem prioridades seguindo a política do deadline monotônico (DM, "deadline monotonic", [LEU 82], [AUD 93a]). O deadline monotônico é uma forma simples e rápida de atribuir prioridades. Em sistemas de tarefas com relações de precedência o DM não é ótimo, como mostra o exemplo apresentado em [AUD 93d]. Entretanto, DM é ótimo na classe das políticas que geram prioridades decrescentes dentro dos grafos de precedência ([HAR 94]). Além disto, DM pode ser considerado uma boa heurística de propósito geral.

Existem na literatura testes para avaliar a escalonabilidade de conjuntos de tarefas independentes, escalonadas segundo políticas de prioridade fixa. Alguns destes testes suportam tarefas que apresentam *jitter* de liberação, tais como [AUD 93c] e [TIN 94a]. Estes dois trabalhos em particular analisam a escalonabilidade de cada tarefa calculando seu tempo máximo de resposta e então comparando este valor com o respectivo deadline.

Neste trabalho vamos desenvolver regras de transformação que, a partir do conjunto original de tarefas, criam conjuntos equivalentes de tarefas independentes com *jitter* de liberação. Estes novos conjuntos de tarefas independentes são então usados para avaliar a escalonabilidade do conjunto original de tarefas.

Esta é uma abordagem bastante usual. Por exemplo, já em [CHE 90a] é descrito um método que transforma um conjunto de tarefas dependentes em um conjunto de tarefas independentes. Esta transformação é feita de forma que uma escala válida pode ser obtida para o conjunto independente se e somente se uma escala válida puder ser obtida para o conjunto dependente. Esta mesma técnica é aplicada em [SUN 94].

Na verdade, neste trabalho vamos criar um conjunto equivalente para as análises de cada tarefa T_i presente no conjunto original. As regras de transformação consideram as características temporais da tarefa T_i em questão. O resultado é um conjunto equivalente tal que: o tempo máximo de resposta da tarefa T_i dentro do novo conjunto é maior ou igual ao tempo máximo de resposta desta mesma tarefa dentro do conjunto original. Isto significa que o teste de escalonabilidade resultante da aplicação das regras de transformação será suficiente mas não necessário.

Uma vez calculado o tempo máximo de resposta da tarefa, podemos avaliar sua escalonabilidade simplesmente comparando este valor com o seu deadline. Assim, teremos

que: se a tarefa em questão for escalonável dentro do novo conjunto, então esta mesma tarefa será escalonável dentro do conjunto original.

A análise da escalonabilidade de cada tarefa do conjunto original de tarefas é então resolvida em três etapas:

Para cada tarefa do conjunto original faça
1) Aplica as regras de transformação para criar um conjunto equivalente;
2) Calcula o tempo máximo de resposta da tarefa no conjunto equivalente;
3) Compara o tempo máximo de resposta calculado com o deadline da tarefa.

4.3.1 Atribuição de Prioridades

Quando tarefas com relações de precedência são distribuídas em vários processadores, pode ocorrer uma situação de dependência mútua entre o cálculo do *jitter* de liberação de uma tarefa e o tempo máximo de resposta de outra tarefa. Este problema foi identificado antes em [TIN 94c] e solucionado através do cálculo simultâneo do tempo máximo de resposta de todas as tarefas, através de sucessivas iterações. Inicialmente é suposto um *jitter* de liberação zero para todas as tarefas e os respectivos tempos de resposta são calculados. O *jitter* de liberação de cada tarefa é então calculado considerando-se os tempos de respostas obtidos. Usando estes novos valores de *jitter* de liberação, os tempos de respostas são novamente calculados. Este processo continua até que os valores de *jitter* de liberação e tempo de resposta de todas as tarefas sejam coerentes entre si.

Uma outra solução, adotada neste trabalho, é limitar a forma como prioridades são atribuídas às tarefas. O problema pode ser eliminado se a política de atribuição de prioridades respeitar dois requisitos:

- As prioridades locais das tarefas são retiradas de uma ordenação global, a qual inclui todas as tarefas do sistema. Embora para efeitos de escalonamento em cada processador apenas a ordenação das tarefas locais seja relevante, vamos supor que esta ordenação local é coerente com uma ordenação global das tarefas da aplicação. Observe que podemos obter a ordenação local para cada processador através de simples inspeção da ordenação global.

- Dentro da ordenação global, prioridades são decrescentes ao longo das relações de precedência. Em outras palavras, se T_i precede direta ou indiretamente T_j então T_i possui prioridade maior que T_j. Esta restrição é razoável, na medida que privilegia as tarefas no início da atividade, cujo tempo de resposta acaba afetando todas as tarefas sucessoras.

Uma política de atribuição de prioridades que é simples e capaz de atender aos dois requisitos colocados é o Deadline Monotônico (DM, "deadline monotonic", [LEU 82], [AUD 93a]). Neste trabalho vamos considerar que as tarefas da aplicação recebem uma prioridade única em toda a aplicação segundo o DM. Tarefas com deadline menor recebem uma prioridade maior. Esta prioridade única vai resultar em uma ordenação global das tarefas. Embora não seja uma política ótima em sistemas de tarefas com relações de precedência, DM é capaz de proporcionar bons resultados de uma forma simples.

Para satisfazer o requisito de prioridades decrescentes ao longo das relações de precedência será necessário tomar dois cuidados:

- Se a tarefa T_i é predecessora direta da tarefa T_j, então será necessário termos $D_i \leq D_j$, onde D_i e D_j são os respectivos deadlines. Esta premissa é bastante razoável. Não tem sentido T_j possuir um deadline menor que T_i uma vez que T_j somente inicia sua execução após o término de T_i. Mesmo que no projeto original tenhamos $D_i > D_j$, podemos transformar o deadline de T_i fazendo $D'_i = D_j - M_j$, onde M_j é o tempo máximo de execução da parte obrigatória de T_j. O novo sistema não é menos escalonável que o original, pois para que T_j cumpra o deadline D_j é necessário que T_i conclua sua execução até D'_i.
- Caso duas tarefas T_i e T_j possuam deadlines iguais, o DM não especifica qual delas receberá a prioridade maior. Vamos arbitrar que, caso nesta situação exista uma relação de precedência entre elas, a tarefa predecessora recebe prioridade maior. Se não existir relação de precedência entre elas, qualquer uma pode ser escolhida. Observe que este cuidado não viola a política DM, mas estabelece um critério onde o DM aceita qualquer solução.

Combinando DM com estes dois cuidados, temos que as prioridades das tarefas geram uma ordenação global e são sempre decrescentes ao longo das relações de precedência. Um efeito colateral desta política de prioridades é um efeito simplificador, como mostrado antes em [HAR 94] para tarefas com relações de precedência e liberação dinâmica em monoprocessador.

4.4 Formulação do Problema

Será empregado o modelo de tarefas básico descrito no capítulo 3. As tarefas recebem individualmente uma prioridade única na aplicação, conforme a política do deadline monotônico. Além disto, é suposto que os deadlines crescem ao longo das relações de precedência. Em outras palavras, se $T_i \in Pred(T_j)$ então $D_i < D_j$ e $\rho(T_i) > \rho(T_j)$, ou seja, a prioridade de T_i será maior que a prioridade de T_j. Sem perda de generalidade vamos supor que as tarefas foram numeradas na ordem decrescente de suas prioridades, de maneira que se $i<j$ então $\rho(T_i) > \rho(T_j)$, ou seja, T_i possui prioridade maior que T_j.

Vamos aqui introduzir a notação $H(T_i)$ para denotar o processador onde a tarefa T_i foi alocada. A notação $\rho(A_x)$ será empregada para denotar a prioridade da tarefa de A_x com mais alta prioridade, ou seja,

$$\rho(A_x) = \max_{\forall T_j . T_j \in A_x} \rho(T_j).$$

Finalmente, a notação R_i^A será usada para denotar o tempo máximo de resposta da tarefa T_i dentro da aplicação **A**. Por exemplo, $R_i^A = R_i^B$ significa que o tempo máximo de resposta de T_i na aplicação **A** é igual ao seu tempo máximo de resposta na aplicação **B**.

Dada uma aplicação com as características descritas acima, é necessário determinar se todas as tarefas sempre serão concluidas antes do respectivo deadline. Para tanto, vamos determinar qual o tempo máximo de resposta para cada tarefa e comparar com o respectivo deadline. A próxima seção descreve regras de transformação que permitem a criação de um sistema equivalente onde as tarefas não possuem relações de precedência. Então, a partir

desse sistema equivalente, será possível aplicar técnicas existentes na literatura para o cálculo do tempo máximo de resposta quando tarefas são independentes.

4.5 Análise da Escalonabilidade

Nesta seção vamos desenvolver regras de transformação que criam um conjunto de tarefas independentes com *jitter* de liberação que é equivalente ao conjunto original de tarefas com relações de precedência. Neste trabalho em particular, algumas destas regras criam um sistema equivalente próximo do original, no sentido que ele poderá ser mais pessimista que o conjunto original. Desta forma, o teste de escalonabilidade como um todo será suficiente mas não necessário. Cada uma destas regras de transformação será apresentada na forma de um teorema.

Os teoremas 1 a 5 tratam da situação onde uma tarefa T_i, única tarefa da atividade A_x, recebe interferência de tarefas pertencentes a outras atividades. Inicialmente, vamos considerar a interferência sobre a tarefa T_i causada por uma atividade A_y que é completamente local, ou seja, todas as tarefas de A_y são executadas no mesmo processador onde T_i é executado. Neste caso, considerando que as prioridades são decrescentes ao longo das relações de precedência, temos 3 situações: todas as tarefas de A_y possuem prioridade menor que $\rho(T_i)$; todas as tarefas de A_y possuem prioridade maior que $\rho(T_i)$; enquanto algumas tarefas de A_y possuem prioridade maior, outras possuem prioridade menor que $\rho(T_i)$. Estas três situações originam, respectivamente, os teoremas 1, 2 e 3.

O teorema 1 mostra que a atividade A_y, completamente local a $H(T_i)$, pode ser ignorada quando todas as suas tarefas possuem prioridade menor que $\rho(T_i)$.

Teorema 1

Seja **A** uma aplicação onde $A_x \in A$ e $A_y \in A$ são atividades tais que $A_x \neq A_y$. Seja T_i a única tarefa da atividade A_x. Suponha ainda que

$$\forall T_j \in A_y \,.\, H(T_j)=H(T_i) \wedge \rho(T_j)<\rho(T_i)$$

e que tenhamos uma aplicação B tal que $B = A - \{A_y\}$.

Nestas condições, temos que $R_i^A = R_i^B$.

Prova

Considere a tarefa T_j, onde $T_j \in A_y$. Sabemos pelas premissas que $\rho(T_j)<\rho(T_i)$ e, portanto, T_j não interfere em T_i. Pela forma como prioridades são atribuídas, sabemos que $\forall T_k, T_k \in Suc(T_j) \Rightarrow \rho(T_k)<\rho(T_j)<\rho(T_i)$ e, portanto, T_k também não interfere na execução de T_i. Logo, T_j não consegue causar interferência direta sobre T_i e somente consegue afetar, através de relações de precedência, tarefas que não causam interferência sobre T_i. Sua remoção não altera o tempo máximo de resposta de T_i.

Repetindo este argumento para cada tarefa T_j que pertence a A_y, podemos eliminar completamente a atividade A_y. □

Neste ponto vamos introduzir a notação $\#dPred(T_i)$ para representar a cardinalidade do conjunto $dPred(T_i)$ dos predecessores diretos da tarefa T_i. De forma similar, vamos usar

#dSuc(T_i), #iPred(T_i) e #iSuc(T_i) para representar respectivamente o número de sucessores diretos, predecessores indiretos e sucessores indiretos da tarefa T_i.

O teorema 2 mostra que a atividade A_y, completamente local a $H(T_i)$, pode ser reduzida a uma única tarefa equivalente quando todas as suas tarefas possuem prioridade maior que $\rho(T_i)$.

Teorema 2

Seja A uma aplicação onde $A_x \in A$ e $A_y \in A$ são atividades tais que $A_x \neq A_y$. Considere a atividade A_x formada por uma única tarefa T_i e a atividade A_y caracterizada por $\forall T_j \in A_y$. $H(T_j)=H(T_i) \wedge \rho(T_j)>\rho(T_i)$ e pelo fato de existir exatamente apenas uma tarefa T_k tal que $T_k \in A_y \wedge \#dPred(T_k)=0$.

Finalmente, seja $B = (A-\{A_y\}) \cup \{A_e\}$ uma aplicação onde A_e é uma atividade com período idêntico ao da atividade A_y, $P_e=P_y$, composta por uma única tarefa T_e tal que:

$J_e = J_k$,
$H(T_e) = H(T_i)$,
$\rho(T_e)=\rho(A_y)$,

$$Ce = \sum_{\forall Tj.Tj \in Ay} Cj \ .$$

Nestas condições, temos que $R_i^A = R_i^B$.

Prova

Sem perda de generalidade, vamos supor que o tempo máximo de resposta R_i^A da tarefa T_i na aplicação **A** ocorre quando T_i é liberada no instante t e concluída no instante t'.

Partindo da aplicação **A**, vamos substituir todas as chegadas da atividade A_y por chegadas de T_e, fazendo com que T_e seja liberada exatamente no mesmo instante de cada liberação de T_k, a tarefa inicial de A_y. Isto é possível pois T_e possui o mesmo período e o mesmo *jitter* de liberação máximo de T_k. Esta substituição corresponde a transformar a aplicação **A** na aplicação **B**.

A computação total requerida pela aplicação **B**, que inclui T_e, no intervalo [t,t') será a mesma requerida pela aplicação **A**, que inclui A_y. Logo, a liberação de T_i que ocorreu no instante t será novamente concluida no instante t'. Temos aqui uma escala de execução possível para a aplicação **B** onde T_i apresenta um tempo de resposta igual ao tempo máximo de resposta de T_i na aplicação **A**. Logo, podemos afirmar que $R_i^A \leq R_i^B$, ou seja, o tempo máximo de resposta de T_i em **B** é, no mínimo, igual ao tempo máximo de resposta de T_i em **A**.

Podemos agora partir de R_i^B e repetir o mesmo raciocínio. Desta vez, toda liberação de T_e é substituida por uma liberação de A_y, resultando na transformação da aplicação **B** na aplicação **A** e provando que $R_i^A \geq R_i^B$.

Uma vez que $R_i^A \leq R_i^B$ e $R_i^A \geq R_i^B$, temos que $R_i^A = R_i^B$.

□

O teorema 3 mostra que a atividade A_y, completamente local a $H(T_i)$, pode ser reduzida a uma única tarefa equivalente quando algumas das tarefas de A_y possuem prioridade maior enquanto outras possuem prioridade menor que $\rho(T_i)$. Esta tarefa equivalente será capaz de interferir em T_i uma única vez, pois suas tarefas de prioridade mais baixa vão fazer com que a atividade seja suspensa até que T_i esteja concluída.

Teorema 3

Seja **A** uma aplicação que inclui as atividades A_x e A_y, $A_x \neq A_y$. A atividade A_x é formada por uma única tarefa T_i. A atividade A_y , formada de tal sorte que

$\forall T_j \in A_y \,.\, R_j^A \leq P_y \wedge H(T_j)=H(T_i)$,

é caracterizada também por

$\exists T_j \in A_y \,.\, \rho(T_j)>\rho(T_i) \;\wedge\; \exists T_j \in A_y \,.\, \rho(T_j)<\rho(T_i)$,

e pelo fato de existir exatamente apenas uma tarefa T_k tal que $T_k \in A_y \;\wedge\; \#dPred(T_k)=0$.

Finalmente, seja $B = (A-\{A_y\}) \cup \{A_e\}$ uma aplicação onde A_e é uma atividade composta por uma única tarefa T_e tal que:

$P_e >> P_i$,

$J_e = J_k$,

$H(T_e) = H(T_i)$,

$\rho(T_e) = \rho(A_y)$,

$$Ce = \sum_{\forall Tj.Tj \in Ay \wedge \rho(Tj)>\rho(Ti)} C_j \quad .$$

Nestas condições, temos que $R_i^A = R_i^B$.

Prova

Sem perda de generalidade, vamos supor que o tempo máximo de resposta R_i^A da tarefa T_i na aplicação **A** ocorre quando T_i é liberada no instante **t** e concluída no instante **t'**.

Uma vez que todas as tarefas de A_y possuem um tempo máximo de resposta menor ou igual ao período, temos que as tarefas da atividade A_y que possuem prioridade maior que $\rho(T_i)$ somente podem ser liberadas uma única vez dentro do intervalo [t,t'). Isto ocorre pois antes de uma segunda liberação é necessário executar as tarefas de A_y com prioridade menor que $\rho(T_i)$, o que só acontece depois do instante **t'**. Pode-se dizer que as tarefas finais de A_y, com menor prioridade, trancam a atividade até que T_i tenha concluído.

Partindo da aplicação **A**, vamos substituir a única liberação da atividade A_y dentro do intervalo [t,t') por uma liberação de T_e, fazendo com que T_e seja liberada exatamente no mesmo instante da liberação de T_k, a tarefa inicial de A_y. Esta substituição corresponde a transformar a aplicação **A** na aplicação **B** durante o intervalo [t,t'). Em função de seu período ser muito maior que P_i, a tarefa T_e também só pode ocorrer uma única vez durante um período da tarefa T_i na aplicação **B**.

A computação total requerida pela aplicação **B**, que inclui T_e, no intervalo [t,t') será a mesma requerida pela aplicação **A**, que inclui A_y. Logo, a liberação de T_i que ocorreu no instante t será novamente concluida no instante t'. Temos aqui uma escala de execução possível para a aplicação **B** onde T_i apresenta um tempo de resposta igual ao tempo

máximo de resposta de T_i na aplicação **A**. Logo, podemos afirmar que $R_i^A \leq R_i^B$, ou seja, o tempo máximo de resposta de T_i em **B** é, no mínimo, igual ao tempo máximo de resposta de T_i em **A**.

Podemos agora partir de R_i^B e repetir o mesmo raciocínio. Desta vez, a liberação de T_e é substituida por uma liberação de A_y, resultando na transformação da aplicação **B** na aplicação **A** e provando que $R_i^A \geq R_i^B$. Uma vez que $R_i^A \leq R_i^B$ e $R_i^A \geq R_i^B$, temos então que $R_i^A = R_i^B$. □

O teorema 3 deixa de ser válido quando alguma tarefa de A_y possui um tempo máximo de resposta maior que o período da atividade. Considere como exemplo o sistema mostrado na figura 4.1, o qual é composto por três tarefas. A figura 4.2 mostra a linha de tempo quando as duas atividades são liberadas simultaneamente no instante zero. Inicialmente T_1 é executada até sua conclusão no instante 2. Neste momento T_3 é liberada mas como T_2 possui prioridade superior é T_2 que passa a ser executada. No instante 10 de tempo T_2 já executou durante 8 unidades de tempo, mas então ocorre uma segunda liberação de T_1, pois o período de sua atividade é 10. Observe que o tempo de resposta de T_3 será maior que o seu período. Quando a segunda liberação de T_1 é concluida no instante 12 o processador volta a executar T_2, que será concluida no instante 14.

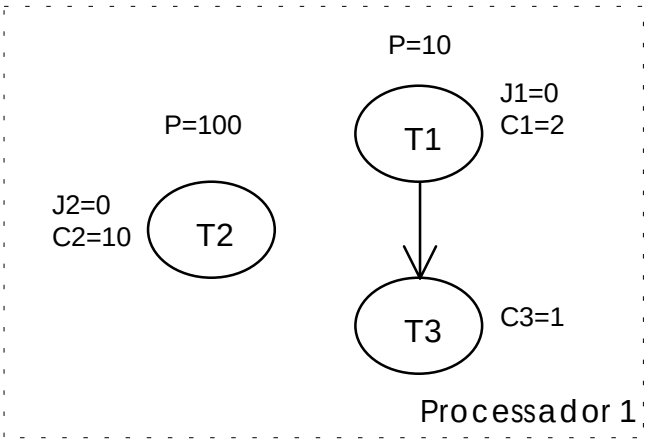

Figura 4.1 - Aplicação hipotética composta por 3 tarefas em um processador.

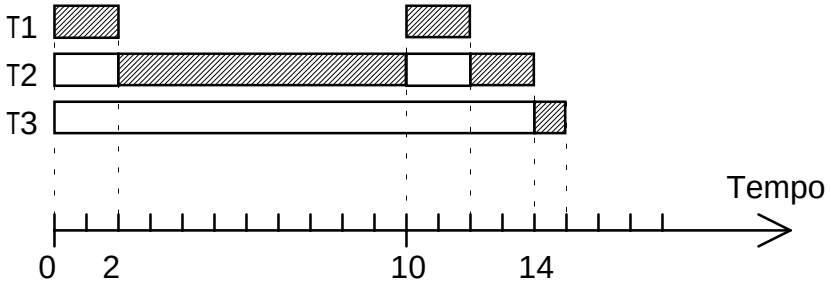

Figura 4.2 - Linha de tempo possível para a aplicação da figura 4.1.

A linha de tempo mostrada na figura 4.2 resultou em um tempo de resposta igual a 14 para a tarefa T_2. Caso o teorema 3 fosse aplicado teriamos uma única interferência de T_1 sobre T_2, resultando em um tempo máximo de resposta igual a 12. Mas o teorema 3 não

pode ser aplicado neste caso pois a tarefa T_3 apresenta um tempo máximo de resposta maior que o seu período.

Neste ponto vamos definir $\delta_{i,j}$ como o atraso associado com o envio de uma mensagem da tarefa T_i para a tarefa T_j. Temos então:

$\delta_{i,j} = \Delta$ caso $H(T_i) \neq H(T_j)$, e
$\delta_{i,j} = 0$ caso $H(T_i) = H(T_j)$.

Os teoremas 4 e 5 tratam de atividades distribuídas que geram interferência sobre uma tarefa independente T_i. Eles descrevem transformações capazes de eliminar as dependências remotas destas atividades distribuídas de modo que tarefas em $H(T_i)$ passam a não depender de tarefas que são executadas em outros processadores. Desta forma, a interferência sobre T_i poderá ser analisada em um contexto local.

O teorema 4 considera uma tarefa T_j executada no mesmo processador que a tarefa T_i e que possui diversos predecessores diretos. Entre as diversas tarefas predecessoras de T_j, a tarefa T_k é aquela capaz de gerar o maior atraso possível na liberação de T_j. É mostrado que a interferência de T_j sobre T_i não é reduzida se transformarmos a aplicação de tal forma que T_j passa a ter, como sua única predecessora direta, a tarefa T_k.

Teorema 4

Seja A uma aplicação que inclui as atividades A_x e A_y, com $A_x \neq A_y$. Considere a atividade A_x formada por uma tarefa única T_i e a atividade A_y tal que

$\exists T_j \in A_y \,.\, H(T_i) = H(T_j) \wedge \#dPred(T_j) \geq 2$.

Seja ainda $T_k \in A_y$ uma tarefa tal que $T_k \in dPred(T_j)$ e

$$R_k^A + \delta_{k,j} = \max_{\forall T_l \in dPred(T_j)} (R_l^A + \delta_{l,j}) .$$

Finalmente, seja $B = [\, A - \{ A_y \} \,] \cup \{ A_e \}$ uma aplicação onde A_e é uma atividade composta pelas mesmas tarefas, relações de precedência e período da atividade A_y, com exceção de T_j que passa a ter $dPred(T_j) = \{ T_k \}$.

Nestas condições, temos que $R_i^A \leq R_i^B$.

Prova

Temos que **B** é a aplicação resultante da eliminação de todas as relações de precedência onde T_j aparece como sucessor, com exceção daquela onde T_k é a tarefa predecessora.

Vamos definir Γ_i^A como o conjunto de todas as escalas de execução possíveis da aplicação **A** no processador onde foi alocada a tarefa T_i. Observe que R_i^A é definida pela escala de execução pertencente a Γ_i^A onde existe a maior distância entre a chegada e a conclusão de T_i.

Uma relação de precedência cria *jitter* de liberação para a tarefa sucessora ao mesmo tempo que reduz o conjunto das escalas de execução possíveis nos processadores.

Não é possível, por exemplo, a tarefa sucessora executar antes da tarefa predecessora dentro do mesmo período da atividade.

No caso específico de T_j na aplicação **A**, o seu *jitter* de liberação máximo é dado por $R_k^A + \delta_{k,j}$. Como a relação de precedência entre T_k e T_j é mantida na aplicação **B**, o mesmo acontece com o *jitter* de liberação máximo de T_j em **B**. A única consequência da remoção das demais relações de precedência é a ampliação do conjunto das escalas de execução possíveis, ou seja, $\Gamma_i^A \subseteq \Gamma_i^B$.

Como temos $\Gamma_i^A \subseteq \Gamma_i^B$, qualquer escala de execução possível em Γ_i^A também será possível em Γ_i^B, inclusive a escala de execução que resultou em R_i^A. Entretanto, nada podemos concluir sobre a existência ou não em Γ_i^B de escalas de execução que resultem em um tempo de resposta maior para T_i. Logo, temos que $R_i^A \leq R_i^B$.

□

O teorema 5 considera uma tarefa T_j executada no mesmo processador que a tarefa T_i e que possui um único predecessor direto. É mostrado que a interferência de T_j sobre T_i não é reduzida se transformarmos a aplicação de tal forma que T_j passa a não ter predecessores diretos. O efeito da relação de precedência eliminada é transformado em um *jitter* de liberação da tarefa T_j.

Teorema 5

Seja A uma aplicação e A_x e A_y atividades tais que $A_x \neq A_y$, $A_x \in A$, $A_y \in A$. Considere a atividade A_x composta por uma tarefa única T_i e a atividade A_y caracterizada por $\exists T_j \in A_y$, $\exists T_k \in A_y$. $H(T_i)=H(T_j) \wedge H(T_j) \neq H(T_k) \wedge dPred(T_j)=\{T_k\}$.

Finalmente, seja $B = (A - \{ A_y \}) \cup \{ A_e \}$ uma aplicação onde A_e é uma atividade composta pelas mesmas tarefas, relações de precedência e período da atividade A_y, com exceção de T_j que passa a ter $dPred(T_j)=\{ \}$ e $J_j = R_k^A + \delta_{k,j}$.

Nestas condições, temos que $R_i^A \leq R_i^B$.

Prova

Temos que **B** é a aplicação resultante da eliminação da única relação de precedência onde T_j aparece como tarefa sucessora. Vamos novamente definir Γ_i^A como o conjunto de todas as escalas de execução possíveis da aplicação **A** no processador onde foi alocada a tarefa T_i. Observe que R_i^A é definida pela escala de execução pertencente a Γ_i^A onde existe a maior distância entre a chegada e a conclusão de T_i.

Uma relação de precedência cria *jitter* de liberação para a tarefa sucessora ao mesmo tempo que reduz o conjunto das escalas de execução possíveis nos processadores. No caso específico da tarefa T_j, o seu *jitter* de liberação máximo na aplicação **A** é dado por $R_k^A + \delta_{k,j}$. Este *jitter* de liberação máximo, oriundo da relação de precedência em **A**, é mantido na aplicação **B**. A única consequência da remoção da relação de precedência é a ampliação do conjunto das escalas de execução possíveis, ou seja, $\Gamma_i^A \subseteq \Gamma_i^B$.

Como temos $\Gamma_i^A \subseteq \Gamma_i^B$, qualquer escala de execução possível em Γ_i^A também será possível em Γ_i^B, inclusive a escala de execução que resultou em R_i^A. Entretanto, nada

podemos concluir sobre a existência ou não em Γ_i^B de escalas de execução que resultem em um tempo de resposta maior para T_i. Logo, temos que $R_i^A \leq R_i^B$.

□

Os teoremas 6 a 9 tratam da situação onde uma tarefa T_i, sujeita às relações de precedência em uma atividade A_X, recebe interferências de tarefas pertencentes a outras atividades. Estes teoremas aplicam transformações sobre a própria atividade da tarefa T_i, até o ponto de tornar T_i uma tarefa independente.

O teorema 6 mostra que é possível eliminar as relações de precedência que ligam a tarefa T_i às demais tarefas da atividade A_X sem afetar o tempo de resposta de T_i. Isto é possível quando a tarefa T_i não possui predecessores.

Teorema 6

Seja A uma aplicação e $A_X \in A$ uma atividade. Seja $T_i \in A_X$ uma tarefa tal que $dPred(T_i)=\{\ \}$. Finalmente, seja $B = (A-\{A_X\}) \cup \{A_i, A_e\}$ uma aplicação onde A_i é uma atividade com período $P_i=P_X$, formada exclusivamente pela tarefa T_i e A_e é uma atividade composta pelas mesmas tarefas e relações de precedência da atividade A_X, com exceção da tarefa T_i que é excluída e do período que será $P_e >> P_X$.

Nestas condições, temos que:

[I] Caso $R_i^A \leq P_X$ então $R_i^A = R_i^B$;

[II] Caso $R_i^A > P_X$ então $R_i^A \geq R_i^B > P_X$.

Prova - Caso [I]

Como $R_i^A \leq P_X$, durante uma execução de T_i na aplicação **A** as demais tarefas de A_X poderão ser liberadas no máximo uma única vez. Uma segunda liberação implica em uma segunda ocorrência de A_X e, portanto, uma segunda chegada de T_i. Na aplicação **B**, o período de A_e garante que suas tarefas poderão ser liberadas no máximo uma única vez durante uma execução de T_i. Logo, a interferência causada pelas demais tarefas de A_X sobre T_i em **A** será igual a causada pelas tarefas de A_e sobre T_i em **B**.

Ao mesmo tempo, eliminar as relações de precedência nas quais T_i aparece como predecessor não afeta a interferência sobre Ti causada por tarefas que pertencem a outras atividades e possuem prioridade maior que T_i.

Dado que as diferenças entre **A** e **B** não afetam a interferência sofrida por T_i, temos $R_i^A = R_i^B$.

Prova - Caso [II]

Uma vez que $R_i^A > P_X$, teremos uma segunda chegada de A_X antes da conclusão de T_i. Esta segunda liberação das tarefas iniciais de A_X vai causar mais interferência sobre T_i e aumentar o próprio valor de R_i^A. Entretanto, para que esta segunda chegada de A_X ocorra antes da conclusão de T_i, é necessário termos $R_i^A > P_X$ mesmo sem considerar a interferência da segunda chegada de A_X.

Logo, caso tenhamos $R_i^A > P_X$ quando a interferência da segunda chegada de A_X é considerada, então teremos $R_i^A > P_X$ mesmo quando a interferência da segunda chegada de

A_X não é considerada. Como a única fonte de interferência eliminada na aplicação **B** em relação à aplicação **A** foram as novas chegadas da atividade A_X, temos então que:
se $R_i^A > P_X$ então $R_i^A \geq R_i^B > P_X$.

□

A transformação realizada pelo teorema 6 é tal que, caso T_i possua na aplicação **A** um tempo máximo de resposta menor ou igual ao seu período, então o seu tempo máximo de resposta na aplicação **B** será o mesmo da aplicação **A**. Caso T_i possua na aplicação **A** um tempo máximo de resposta maior que o seu período, então o seu tempo máximo de resposta na aplicação **B** poderá ser menor que em **A**, mas ainda assim maior que o seu período.

Como temos sempre que o deadline é menor ou igual ao período, a transformação do teorema 6 é tal que: se T_i é escalonável em **A**, continua escalonável em **B**; se T_i não é escalonável em **A**, continua não escalonável em **B**, porém com um tempo máximo de resposta possivelmente menor. Neste último caso a transformação é otimista, mas não a ponto de transformar uma tarefa não escalonável em tarefa escalonável.

O teorema 7 considera a situação onde uma tarefa T_i possui uma única tarefa predecessora T_k, alocada ao mesmo processador que a tarefa T_i. Neste caso, é possível unificar as duas tarefas, criando uma única tarefa equivalente.

Teorema 7

Seja **A** uma aplicação e $A_X \in A$ uma atividade. Seja $T_i \in A_X$ uma tarefa tal que $dPred(T_i)=\{T_k\}$, onde $H(T_i)=H(T_k)$.

Finalmente, seja $B = (A-\{A_X\}) \cup \{A_e\}$ uma aplicação onde A_e é uma atividade composta pelas mesmas tarefas, relações de precedência e período da atividade A_X, com exceção das tarefas T_k e T_i que são substituidas pela tarefa T_e, com

$H(T_e) = H(T_i)$,
$\rho(T_e) = \rho(T_i)$,
$J_e = J_k$,
$C_e = C_k + C_i$,
$dPred(T_e) = dPred(T_k)$,
$dSuc(T_e) = dSuc(T_i)$.

Nestas condições, temos que $R_i^A = R_e^B$.

Prova

Vamos transformar passo a passo a aplicação **A** na aplicação **B**, mostrando que o tempo máximo de resposta de T_i em **A** não é afetado pelas transformações e será igual ao tempo máximo de resposta de T_e em **B**.

É possível a existência de tarefas em **A** que sucedem T_k mas não sucedem T_i. Estas tarefas possuem prioridade inferior a $\rho(T_k)$, mas poderão também possuir prioridade superior a $\rho(T_i)$ e, em função disto, interferir com T_i. Este fato independe das suas relações de precedência com T_k. Logo, tais relações de precedência podem ser eliminadas sem afetar R_i^A.

Vamos mudar a prioridade de T_k para $\rho(T_i)$. No intervalo de tempo [t,t'), definido pela liberação de T_k e pela conclusão de T_i no pior caso, existe tempo suficiente para a execução completa das tarefas T_k, T_i e de toda tarefa T_j tal que $\rho(T_j)>\rho(T_i) \wedge H(T_j)=H(T_i)$ que tenha sido liberada antes de **t'** mas não concluida antes de **t**. Logo, se mudarmos a prioridade de T_k para o valor $\rho(T_i)$, teremos uma reordenação das tarefas dentro deste intervalo, de tal sorte que T_k será executado mais ao final do intervalo, juntamente com T_i. De qualquer forma, T_i será concluido em **t'** no pior caso.

Vamos agora acrescentar ao conjunto $dPred(T_i)$ todas as tarefas pertencentes ao conjunto $dPred(T_k)$. Esta alteração não altera o momento da liberação de T_i, pois as tarefas $dPred(T_k)$ pertencem originalmente ao conjunto $iPred(T_i)$ e já terão sido concluidas quando T_i é liberado.

Vamos agora eliminar a relação de precedência entre T_k e T_i e fazer com que T_i seja liberada no mesmo instante que T_k. Nessa transformação a tarefa T_i passa a apresentar o mesmo *jitter* de liberação que a tarefa T_k, pois ambas passam a ter os mesmos instantes de chegada e de liberação. Teremos novamente uma reordenação das tarefas dentro do intervalo [t,t'), de tal sorte que T_i poderá executar antes da conclusão de T_k. De qualquer forma, as tarefas T_k e T_i serão ainda concluidas ao final do intervalo [t,t') no pior caso, pois a demanda por processador não foi alterada e prioridades iguais implicam em uma ordem de execução arbitrária.

Uma vez que, após as transformações, T_k e T_i possuem o mesmo instante de liberação, o mesmo *jitter* de liberação máximo, a mesma prioridade, executam no mesmo processador, possuem os mesmos predecessores e os mesmos sucessores, podemos considera-los como se fossem uma única tarefa. A tarefa única equivalente T_e possui um tempo máximo de execução igual a soma dos tempos máximos de execução de T_i e T_k e será concluida no instante t', no pior caso. Temos portanto que $R_i^A = R_e^B$. □

O teorema 8 considera a situação onde uma tarefa T_i possui uma única tarefa predecessora T_k, alocada a um processador diferente daquele onde está a tarefa T_i. Neste caso, é possível eliminar a relação de precedência e substitui-la pela associação de um *jitter* de liberação à tarefa T_i, de forma que o tempo máximo de resposta de T_i é mantido sem alteração.

Teorema 8

Seja **A** uma aplicação e $A_X \in A$ uma atividade. Seja $T_i \in A_X$ uma tarefa tal que $dPred(T_i)=\{T_k\}$, onde $H(T_i)\neq H(T_k)$.

Finalmente, seja $B = (A-\{A_X\})\cup\{A_e\}$ uma aplicação onde A_e é uma atividade composta pelas mesmas tarefas, relações de precedência e período da atividade A_X, com exceção de que a tarefa T_i é substituida por uma tarefa T_e e de que todas as tarefas pertencentes ao conjunto $iPred(T_i)$ em **A** são tais que, por definição, não causam interferência sobre T_e em **B**. A tarefa T_e é definida por:

$H(T_e) = H(T_i)$,
$\rho(T_e) = \rho(T_i)$,
$C_e = C_i$,
$J_e = R_k^A + \Delta$,
$dPred(T_e) = \{\ \}$.

Nestas condições, temos que:

[I] Caso $R_i^A \leq P_x$ então $R_i^A = R_e^B$;
[II] Caso $R_i^A > P_x$ então $R_i^A \geq R_e^B > P_x = P_e$.

Prova - Caso [I]

Uma relação de precedência cria *jitter* de liberação para a tarefa sucessora ao mesmo tempo que reduz o conjunto de escalas de execução possíveis no sistema. Tarefas predecessoras, diretas ou indiretas, atrasam a liberação da tarefa sucessora mas não geram interferência sobre ela. Vamos comparar T_i em **A** com T_e em **B** no que diz respeito ao *jitter* de liberação máximo e as interferências sofridas.

O *jitter* de liberação gerado por T_k sobre T_i em **A** é máximo quando T_k apresentar o seu tempo máximo de resposta e a comunicação entre T_k e T_i apresentar o atraso máximo, isto é, $R_k^A + \Delta$. Pela forma como foi definida, a tarefa T_e apresenta um *jitter* de liberação máximo em **B** que é igual ao *jitter* de liberação máximo de T_i em **A**.

Considerando a interferência sofrida por T_i em **A**, podemos afirmar que no momento que T_i é liberada necessariamente todos os predecessores indiretos já estão concluidos e, como $R_i^A \leq P_x$, não serão liberados novamente antes da conclusão de T_i. Logo, eles não interferem na execução de T_i além de contribuir para o seu *jitter* de liberação. Pela forma como foi definida, a tarefa T_e também não recebe interferência das tarefas que precedem indiretamente T_i em **A**.

Uma vez que os efeitos da relação de precedência entre T_k e T_i em **A** foram compensados pelas características de T_e em **B**, podemos afirmar que $R_i^A = R_e^B$.

Prova - Caso [II]

Uma vez que $R_i^A > P_x$, teremos uma segunda chegada de A_x antes da conclusão de T_i. Esta segunda chegada de A_x resultará em uma segunda liberação dos predecessores indiretos de T_i antes da conclusão de T_i. Esta segunda liberação dos predecessores indiretos de T_i vai causar mais interferência sobre T_i e aumentar o próprio valor de R_i^A. Entretanto, para que esta segunda liberação dos predecessores indiretos de T_i ocorra antes da conclusão de T_i, é necessário termos $R_i^A > P_x$ mesmo sem considerar a interferência causada por tal liberação.

Logo, caso tenhamos $R_i^A > P_x$ quando a interferência dos predecessores indiretos de T_i é considerada, então teremos $R_i^A > P_x$ mesmo quando esta interferência não é considerada. Esta última situação é o que ocorre na aplicação **B**, pois esta é a única fonte de interferência eliminada na transformação. Temos então que se $R_i^A > P_x$ então teremos também $R_i^A \geq R_e^B > P_x$.

□

De forma semelhante ao teorema 6, a transformação realizada pelo teorema 8 é tal que, caso T_i possua na aplicação **A** um tempo máximo de resposta menor ou igual ao seu período, então ele será igual ao tempo máximo de resposta de T_e na aplicação **B**. Caso T_i possua na aplicação **A** um tempo máximo de resposta maior que o seu período, então o tempo máximo de resposta de T_e na aplicação **B** poderá ser menor que R_i^A, mas ainda assim maior que o período da atividade.

Como temos sempre que o deadline é menor ou igual ao período, a transformação do teorema 8 é tal que: se T_i é escalonável em **A**, T_e será escalonável em **B**; se T_i não é escalonável em **A**, Te não será escalonável em **B**, porém apresentará um um tempo máximo de resposta possivelmente menor que T_i em **A**. Novamente, a transformação é otimista, mas não a ponto de transformar uma tarefa não escalonável em tarefa escalonável.

O teorema 9 considera uma tarefa T_i que possui diversos predecessores diretos. Entre as diversas tarefas predecessoras de T_i, a tarefa T_k é aquela capaz de gerar o maior tempo de resposta possível para T_i. É mostrado que o tempo máximo de resposta de T_i não é alterado se transformarmos a aplicação de tal forma que T_i passa a ter, como sua única predecessora direta, a tarefa T_k.

Teorema 9

Seja **A** uma aplicação e A_x uma atividade tal que $A_x \in A$. Seja T_i uma tarefa tal que $T_i \in A_x$ e $\#dPred(T_i) \geq 2$. Seja T_{loc} uma tarefa tal que $T_{loc} \in A_x$, $T_{loc} \in dPred(T_i)$ e ainda $H(T_{loc})=H(T_i)$, onde

$$R_{loc}{}^{A} = \max_{\forall T_l . T_l \in dPred(T_i) \wedge H(T_l)=H(T_i)} R_l{}^{A} .$$

De forma semelhante, seja T_{rem} uma tarefa tal que $T_{rem} \in A_x$, $T_{rem} \in dPred(T_i)$, $H(T_{rem}) \neq H(T_i)$, onde

$$R_{rem}{}^{A} + \Delta = \max_{\forall T_l . T_l \in dPred(T_i) \wedge H(T_l)\neq H(T_i)} R_l{}^{A} + \Delta .$$

Vamos definir a tarefa T_k, predecessora crítica de T_i em **A**, da seguinte forma:

[a] Caso $\forall T_j . T_j \in dPred(T_i) \Rightarrow H(T_j) \neq H(T_i)$ então $T_k = T_{rem}$.

[b] Caso $\forall T_j . T_j \in dPred(T_i) \Rightarrow H(T_j) = H(T_i)$ então $T_k = T_{loc}$.

[c] Caso $R^A_{rem} + \Delta \geq R^A_{loc}$ então $T_k = T_{rem}$.

[d] Caso $R^A_{rem} + \Delta < R^A_{loc} - I^A_{loc}$ então $T_k = T_{loc}$, onde I^A_{loc} é a interferência máxima recebida por T_{loc} em **A**.

Finalmente, seja $B = (A - \{A_x\}) \cup \{A_e\}$ uma aplicação onde A_e é uma atividade composta pelas mesmas tarefas, relações de precedência e período da atividade A_x, com exceção da tarefa T_i que passa a ter $dPred(T_i)=\{ T_k \}$ e de que, se $H(T_k) \neq H(T_i)$ então todas as tarefas $T_j . T_j \in Pred(T_i) \wedge H(T_j)=H(T_i)$ em **A** são tais que, por definição, não causam interferência sobre T_i em **B**.

Nestas condições, temos que:

[I] Caso $R_i{}^A \leq P_x$ então $R_i{}^A = R_i{}^B$;

[II] Caso $R_i{}^A > P_x$ então $R_i{}^A \geq R_i{}^B > P_x = P_e$.

Vamos considerar separadamente cada possibilidade para a definição de T_k.

Caso [a]

No pior caso, a tarefa T_{rem} é a última entre as tarefas predecessoras diretas de T_i a terminar e, portanto, a enviar uma mensagem para T_i. É garantido que, quando T_{rem} termina no pior caso, todas as demais tarefas predecessoras diretas e indiretas de T_i também já terão terminado. Desta forma, mantendo a relação de precedência entre T_{rem} e T_i estaremos mantendo também o tempo máximo de resposta de T_i.

Considerando a interferência sofrida por T_i em **A**, podemos afirmar que, no momento que T_i é liberada, necessariamente todos os predecessores locais já estão concluidos. Como $R_i^A \leq P_x$, eles não serão liberados novamente antes da conclusão de T_i. Logo, eles não interferem na execução de T_i. O mesmo acontece, por definição, na aplicação **B**.

Caso [b]

A tarefa T_{loc}, no pior caso, é a última entre as tarefas predecessoras diretas de T_i a terminar e, portanto, a enviar uma mensagem para T_i. Como todos os predecessores diretos de T_i são locais, é garantido que, quando T_{loc} termina no pior caso, todas as demais tarefas predecessoras diretas de T_i também já terão terminado. Desta forma, mantendo a relação de precedência entre T_{loc} e T_i estaremos mantendo também o tempo máximo de resposta da tarefa T_i.

Caso [c]

Quando a mensagem da tarefa T_{rem} chegar até a tarefa T_i, todas as tarefas predecessoras diretas e indiretas de T_i já estarão concluidas e suas mensagens já terão chegado até T_i. Logo, o tempo máximo de resposta de T_i é definido pela relação de precedência que inclui T_{rem}, podendo as demais relações de precedência serem eliminadas.

Considerando a interferência sofrida por T_i em **A**, podemos afirmar que, no momento que T_i é liberada, necessariamente todos os predecessores locais já estão concluidos. Como $R_i^A \leq P_x$, eles não serão liberados novamente antes da conclusão de T_i. Logo, eles não interferem na execução de T_i. O mesmo acontece, por definição, na aplicação **B**.

Caso [d]

O pior caso de interferência sobre T_i pode ser diferente do pior caso de interferência sobre T_{loc}. Por exemplo, considere uma tarefa com período muito maior que P_x. No pior caso para T_i ela é liberada exatamente no mesmo instante que T_i é liberada, isto é, após a conclusão de T_{loc}. Certamente este não é o pior caso do ponto de vista de T_{loc}. Assim, ao computarmos I^A_{loc} estamos construindo um cenário que é o pior para T_{loc}, mas não é necessariamente o pior para T_i. O fato de termos $R^A_{rem} + \Delta < R^A_{loc}$ significa que a mensagem de T_{rem} chegará antes de T_{loc} terminar no pior cenário para T_{loc}. Entretanto, isto não garante que a mensagem de T_{rem} chegará antes de T_{loc} terminar no pior cenário para T_i. Logo, não basta termos $R^A_{rem} + \Delta < R^A_{loc}$ para definirmos T_{loc} como a precedência crítica de T_i.

Ao supor que $R^A_{rem} + \Delta < R^A_{loc} - I^A_{loc}$, estaremos também supondo que, independentemente dos padrões de interferência sobre T_{loc} e T_i, a mensagem de T_{rem} chegará em T_i antes de T_{loc} terminar. Logo, independentemente dos padrões de interferência, o pior caso para T_i está associado com a conclusão de T_{loc} e não com a mensagem enviada por T_{rem}. Ao preservar a relação de precedência entre T_{loc} e T_i estaremos preservando o pior cenário para T_i e o seu tempo máximo de resposta.

Prova [II]

Mesmo raciocínio empregado no caso [II] dos teoremas 6 e 8.

□

O teorema 9 apresenta o mesmo efeito dos teoremas 6 e 8 com respeito à interpretação dos testes de escalonabilidade. Caso $R_i^A > P_X$ a transformação será otimista, mas não a ponto de transformar uma tarefa não escalonável em tarefa escalonável.

Observe que o enunciado do teorema 9 não define T_k quando:
$R^A_{loc} - I^A_{loc} \leq R^A_{rem} + \Delta < R^A_{loc}$.

Nesta situação, podemos determinar o tempo máximo de resposta de T_i atarvés de um aumento artificial de Δ. O exemplo mostrado na figura 4.3 ilustra esta situação. A aplicação em questão possui duas atividades, uma com três e a outra com uma única tarefa. São utilizados dois processadores. Os períodos, *jitter* máximos de liberação e tempos máximos de computação das tarefas são apresentados na própria figura, assim como as relações de precedência existentes.

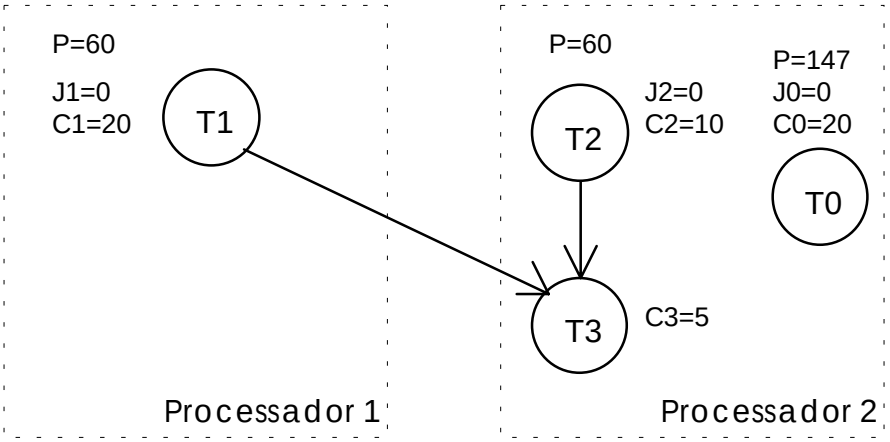

Figura 4.3 - Aplicação hipotética composta por 4 tarefas, Δ=7.

Suponha que o tempo de resposta de T_3 deve ser determinado. Como a tarefa T_3 possui dois predecessores diretos, vamos usar o teorema 9 para identificar o predecessor crítico. Como existe apenas um predecessor direto remoto, temos que $T_{rem}=T_1$. Da mesma forma, como existe um único predecessor direto local, $T_{loc}=T_2$. Temos ainda que

$R^A_{rem}+\Delta=R^A_1+\Delta=20+7=27$,

e que

$I^A_{loc}=I^A_2=20$ e $R^A_{loc}=R^A_2=10+20=30$.

Como

$R^A_{rem}+\Delta < R^A_{loc}$, e

$R^A_{rem}+\Delta \geq R^A_{loc} - I^A_{loc}$, o teorema 9 não define a tarefa crítica T_k.

Neste exemplo pequeno é possível realizar um exame exaustivo das possibilidades. Caso $T_k=T_{rem}=T_1$, então o tempo máximo de resposta de T_3 seria 27+5+20=52. Caso a precedência crítica fosse $T_k=T_{loc}=T_2$, então o tempo máximo de resposta de T_3 seria dado por 10+5+20=35. Claramente neste caso temos que $T_k=T_{rem}$.

Em geral este tipo de análise apresenta complexidade exponencial e não é viável realiza-lo em sistemas com um tamanho mais realista. Nestes casos, vamos aumentar artificialmente o valor de Δ no caso específico da comunicação entre T_1 e T_3 de forma a satisfazer a condição [c] do teorema 9. No caso do exemplo da figura 4.3, se assumirmos o valor Δ=10, teremos que $R^A_{rem}+\Delta = 30 = R^A_{loc}$ e então teremos T_1 como a precedêncnia crítica de T_3.

Este aumento artificial do valor Δ poderá introduzir um certo grau de pessimismo na análise. Isto acontece quando a precedência crítica já é originalmente a precedência remota e o aumento do Δ é desnecessário. Entretanto, para reconhecer esta situação de pessimismo exagerado, seria necessário computar o tempo de resposta de T_i por um e por outro caminho, comparando os dois. Como dito antes, este tipo de análise apresenta complexidade 2^N no pior caso, onde **N** é o número de tarefas da aplicação.

É importante observar que uma única comunicação tem o seu tempo máximo de atraso alterado, isto é, a comunicação entre T_{rem} e T_i. Mesmo assim, esta alteração somente é válida para o cálculo do tempo máximo de resposta de T_i. O cálculo do tempo máximo de resposta das demais tarefas será feito com todas as comunicações apresentando, a princípio, Δ=7. A próxima seção descreve como os teoremas são utilizados dentro de um algoritmo que computa o tempo máximo de resposta de cada tarefa da aplicação.

4.6 Algoritmo Geral

Nesta seção é apresentado um algoritmo capaz de calcular o tempo máximo de resposta de cada tarefa. Uma vez calculado, este tempo pode ser comparado com o respectivo deadline. O teste de escalonabilidade se resume então a uma simples comparação. Mesmo que os deadlines da aplicação estejam associados somente com as tarefas finais de cada atividade, o método empregado exige que o tempo máximo de resposta de todas as tarefas seja calculado.

O princípio básico do algoritmo são sucessivas transformações da aplicação, até que as relações de precedência tenham sido eliminadas. Estas transformações são feitas de tal forma que o tempo máximo de resposta da tarefa T_i na aplicação transformada será maior ou igual ao seu tempo máximo de resposta na aplicação original. As transformações consideram especificamente o efeito das demais tarefas sobre a T_i em questão. Logo, para cada tarefa T_i será necessário aplicar as transformações a partir da aplicação original.

O algoritmo é na verdade composto de três partes. A parte geral, que controla sua execução, é um laço repetido para cada tarefa T_i da aplicação. Inicialmente, as relações de precedência na atividade a qual pertence T_i são eliminadas e o cálculo do tempo máximo de resposta de T_i é reduzido ao problema do cálculo do tempo máximo de resposta de uma tarefa independente que recebe interferência de outras atividades. Em seguida, o tempo

máximo de resposta desta tarefa independente é calculado. Estas duas etapas são realizadas pelas segunda e terceira partes do algoritmo, respectivamente.

A tabela 4.1 apresenta a parte geral do algoritmo. Ela analisa as tarefas da aplicação na ordem crescente das prioridades. No momento que R_i^A é calculado, já será conhecido o R_j^A de todas as tarefas T_j tal que $\rho(T_j)>\rho(T_i)$. As linhas 2 e 3 são simplesmente chamadas para as segunda e terceira partes do algoritmo, respectivamente.

[1]	Para cada T_i, **i** de **1** até **n**, faça
[2]	Obtém uma aplicação equivalente onde T_i é uma tarefa independente
[3]	Calcula o tempo máximo de resposta de T_i na aplicação equivalente

Tabela 4.1 - Primeira parte do algoritmo: parte geral.

A tabela 4.2 apresenta a segunda parte do algoritmo. A atividade que contém T_i sofre sucessivas transformações até que seja obtida uma tarefa independente cujo tempo de resposta será maior ou igual ao tempo de resposta de T_i na aplicação original. Basicamente, a tarefa T_i é aglutinada com os seus predecessores que conseguem gerar o maior tempo de resposta possível. Esta aglutinação acontece até que seja encontrada uma tarefa inicial da atividade ou uma tarefa que é executada em outro processador.

Caso T_i possua um único predecessor direto executado no mesmo processador, T_i e seu predecessor são aglutinados, formando uma nova tarefa na aplicação transformada (linhas 5 e 6). Caso T_i possua um único predecessor direto que executa em outro processador, a relação de precedência é transformada em *jitter* de liberação sofrido pela nova tarefa (linhas 7 e 8). Caso T_i possua vários predecessores diretos, apenas aquele capaz de retardar ao máximo a execução de T_i é mandido. As demais relações de precedência são eliminadas (linhas 9 e 10).

É importante observar que o laço cujo corpo ocupa as linhas de 5 a 10 é executado até que T_i não possua mais predecessores (linha 4). Neste momento, a linha 11 elimina todas as relações de precedência restantes que ligam T_i às suas tarefas sucessoras na atividade. Isto é possível pois estas relações de precedência não afetam o tempo máximo de resposta de T_i.

[4]	Enquanto $\#dPred(T_i) > 0$
[5]	Caso $\#dPred(T_i) = 1 \ \wedge \ H(T_i) = H(T_k)$, $T_k \in dPred(T_i)$
[6]	Aplica teorema 7, aglutinando T_k e T_i
[7]	Caso $\#dPred(T_i) = 1 \ \wedge \ H(T_i) \neq H(T_k)$, $T_k \in dPred(T_i)$
[8]	Aplica teorema 8, reduzindo para $\#dPred(T_i)=0$
[9]	Caso $\#dPred(T_i) > 1$
[10]	Aplica teorema 9, reduzindo para $\#dPred(T_i)=1$
[11]	Aplica teorema 6, separando T_i do resto de A_x

Tabela 4.2 - Segunda parte do algoritmo: torna T_i uma tarefa independente.

Ao final da segunda parte temos uma tarefa T_i que é independente. Vamos agora aplicar o algoritmo que calcula o tempo de resposta de uma tarefa independente quando no meio de atividades distribuídas. Isto é feito como mostrado na tabela 4.3.

A terceira parte do algoritmo corresponde a um laço que vai eliminar as relações de precedência presentes em todas as atividades que possuem tarefas no mesmo processador que T_i (linhas 12 e 13). Entre estas atividades se inclui o que restou da atividade original de T_i, após a sua remoção.

Cada atividade será decomposta em fragmentos. Cada fragmento é identificado por possuir uma única tarefa inicial, que também define o que é um fragmento. Esta decomposição é feita através da eliminação de relações de precedência duplicadas (linhas 14 a 16) e oriundas de outros processadores (linhas 17 até 19). Cada fragmento pode ser considerado como uma atividade em separado, já que não possui qualquer relação de precedência com outros fragmentos.

[12] Para cada $A_y \in A$, $T_i \notin A_y$, faça
[13] Se $\exists T_j \in T.T_j \in A_y \wedge H(T_j)=H(T_i)$ então

[14] Para cada $T_j \in T.T_j \in A_y \wedge H(T_j)=H(T_i) \wedge \rho(T_j)>\rho(T_i)$
[15] Se #dPred(Tj)≥2 então
[16] Aplica teorema 4, reduzindo para #dPred(Tj)=1

[17] Para cada $T_j \in T.T_j \in A_y \wedge H(T_j)=H(T_i) \wedge \rho(T_j)>\rho(T_i)$
[18] Se $T_k \in dPred(T_j) \wedge H(T_k) \neq H(T_i)$ então
[19] Aplica teorema 5, reduzindo para $\#dPred(T_j)=0$

[20] Para cada fragmento independente A_z de A_y em $H(T_i)$

[21] Caso $\forall T_j \in A_z$, $\rho(T_j)<\rho(T_i)$
[22] Aplica teorema 1, eliminando o fragmento Az
[23] Caso $\forall T_j \in A_z$, $\rho(T_j)>\rho(T_i)$
[24] Aplica teorema 2, reduzindo A_z a uma tarefa única
[25] Outros casos
[26] Aplica teorema 3, reduzindo A_z a uma tarefa única

[27] Calcula o tempo máximo de resposta de T_i

Tabela 4.3 - Terceira parte do algoritmo: Calcula o tempo de resposta.

Finalmente, cada fragmento (linha 20) é reduzido a uma tarefa independente que gera sobre T_i uma interferência equivalente. Caso todas as tarefas do fragmento possuam prioridade inferior a T_i, o fragmento é eliminado (linhas 21 e 22). Caso todas as tarefas do fragmento possuam prioridade superior à T_i, o fragmento é transformado em uma tarefa única equivalente (linhas 23 e 24). Caso algumas tarefas possuam prioridade inferior enquanto outras tarefas do mesmo fragmento possuem prioridade superior à T_i, temos que a atividade também é transformada em uma tarefa única equivalente (linhas 25 e 26). Ao final (linha 27), é aplicado um algoritmo para o cálculo do tempo máximo de resposta de

uma tarefa em sistemas com tarefas independentes. Por exemplo, pode ser usado o algoritmo descrito em [AUD 93c].

Vamos agora examinar os casos particulares que podem ocorrer quando os teoremas 3, 6, 8 e 9 são aplicados. O teorema 3 coloca como premissa que os tempos máximos de resposta de todas as tarefas sejam menores ou iguais ao período da respectiva atividade. Logo, com respeito ao resultado da aplicação do algoritmo, existem duas situações possíveis. Caso esta premissa do teorema 3 se verifique, então todos os tempos máximos de resposta calculados são válidos. Caso algum tempo de resposta calculado seja maior que o período da respectiva atividade, a aplicação não é escalonável na presente alocação e os tempos máximos de resposta calculados devem ser considerados como uma aproximação. De qualquer forma, neste último caso, será necessário alterar a aplicação pois ela não é escalonável na presente forma.

De maneira semelhante, a transformação realizada pelos teoremas 6, 8 e 9 somente é exata quando o tempo máximo de resposta da tarefa em questão for menor ou igual ao período de sua atividade. Temos novamente duas situações possíveis. Caso o tempo máximo de resposta da tarefa em questão seja menor ou igual ao período de sua atividade, o cálculo é válido. Caso o tempo máximo de resposta calculado seja maior que o período da atividade, então a tarefa em questão não é escalonável e o seu tempo máximo de resposta calculado deve ser considerado uma aproximação.

Podemos resumir o efeito dos teoremas 3, 6, 8 e 9 da forma mostrada abaixo, onde R_i representa o tempo máximo de resposta calculado para a tarefa T_i e R_i^A representa o tempo máximo de resposta verdadeiro de T_i na aplicação original:

- Caso $\exists T_i \in A_x . R_i > P_x$ então a aplicação não é considerada escalonável e os tempos máximos de resposta calculados para todas as tarefas são aproximações (pessimistas ou otimistas);

- Caso $\forall T_i \in A_x . R_i \leq P_x \wedge \exists T_j \in A_y . R_j > D_j$ então a aplicação não é considerada escalonável e os tempos máximos de resposta calculados para todas as tarefas são aproximações pessimistas, ou seja, $\forall T_k . R_k \geq R_k^A$;

- Caso $\forall T_i \in A_x . R_i \leq P_x \wedge \forall T_i \in A_x . R_i \leq D_i$ então a aplicação é escalonável e os tempos máximos de resposta calculados para todas as tarefas são aproximações pessimistas, ou seja, $\forall T_k . R_k \geq R_k^A$.

Através de uma simples inspeção do algoritmo apresentado é possível notar que qualquer grafo de precedência pode ser analisado. Considere uma tarefa T_i, pertencente à atividade A_x. A segunda parte do algoritmo, apresentada na tabela 4.2, transforma a situação na qual #dPred(T_i)>1 em uma situação do tipo #dPred(T_i)=1 (teorema 9). Em seguida, caso o predecessor seja remoto, ele é simplesmente eliminado (teorema 8). Caso o predecessor seja local, ele é aglutinado com T_i (teorema 7) e o processo repetido. Ao final, temos uma tarefa sem predecessores que pode ser desvinculada do restante da atividade A_x (teorema 6).

Considere agora uma atividade A_y que possui algumas tarefas que causam interferência sobre T_i. Inicialmente cada tarefa de Ay é transformada de tal forma que passe

a ter um único predecessor (teorema 4). Esta transformação tem o efeito de tornar a atividade A_y um conjunto de fragmentos lineares. As precedências remotas são eliminadas em seguida, sendo transformadas em *jitter* de liberação (teorema 5). Neste momento passam a existir apenas fragmentos lineares e completamente locais. Os fragmentos são analisados um a um. Existem três possibilidades: todas as tarefas do fragmento possuem prioridades inferiores a $\rho(T_i)$, todas as tarefas do fragmento possuem prioridades superiores a $\rho(T_i)$ ou então uma situação mista. O algoritmo possui um tratamento específico para cada um destes três casos (teoremas 1, 2 e 3, respectivamente).

É importante notar que as únicas transformações realizadas pelo algoritmo que introduzem pessimismo na análise são aquelas baseadas nos teoremas 4, 5 e, em alguns casos, teorema 9. Com exceção destes teoremas, todos os demais transformam a aplicação **A** em uma aplicação **B** de tal forma que o tempo máximo de resposta da tarefa T_i em questão permanece igual, isto é, $R_i^A = R_i^B$. Ao mesmo tempo, quando uma análise simples baseada em *jitter* de liberação é aplicada (por exemplo, a análise descrita em [BUR 93]), as situações descritas pelas premissas dos teoremas 4 e 5 são tratadas exatamente da mesma forma que no algoritmo apresentado neste trabalho. No caso da situação associada com o teorema 9, quando o valor Δ é elevado artificialmente, a análise simples considera $T_k=T_{loc}$, mas supõe sempre a ocorrência simultânea do pior caso para T_{loc} e T_i. O pessimismo resultante desta solução é igual ao resultante do acréscimo efetuado no Δ.

Logo, como as únicas fontes de pessimismo presentes em nosso algoritmo estão também presentes na abordagem descrita em [BUR 93], nossa análise não é mais pessimista que aquela. A consequência disto é que qualquer aplicação declarada escalonável pela abordagem simples também será declarada escalonável pelo algoritmo apresentado neste trabalho. O contrário não é verdadeiro. Existem aplicações que são declaradas escalonáveis pelo nosso algoritmo mas não o são por uma análise mais simples. O exemplo da seção 7 ilustra esta situação.

Vamos considerar agora a complexidade do algoritmo apresentado. A primeira parte do algoritmo é um laço cujo corpo será executado **n** vezes. A segunda parte poderá ser executada **n** vezes, quando T_i é a última tarefa de uma atividade linear composta por **n** tarefas. O laço principal da terceira parte (linha 12) poderá ser executado **n-1** vezes, quando não existirem relações de precedência na aplicação. Neste caso, os laços internos (linhas 14, 17 e 20) estão limitados a uma única execução. Em outra situação extrema, T_i sofre interferência de uma atividade que inclui todas as demais tarefas da aplicação. Neste caso o laço externo executa **1** vez e os laços internos estão limitados por **n** (nenhuma atividade poderá possuir mais que **n** tarefas). Vamos supor que a linha 27 possui complexidade **E**. Este valor depende do teste de escalonabilidade para tarefas independentes que for utilizado. O algoritmo como um todo possui complexidade dada por n(n+(n+E)), ou seja, $\mathbf{O(n^2+n.E)}$.

4.7 Exemplo

Nesta seção vamos calcular o tempo de resposta das tarefas pertencentes a uma aplicação hipotética. A figura 4.4 mostra como as tarefas da aplicação foram alocadas e quais são as suas relações de precedência. A tabela 4.4 apresenta os valores de *jitter* de liberação, tempo máximo de computação, período e deadline de cada tarefa.

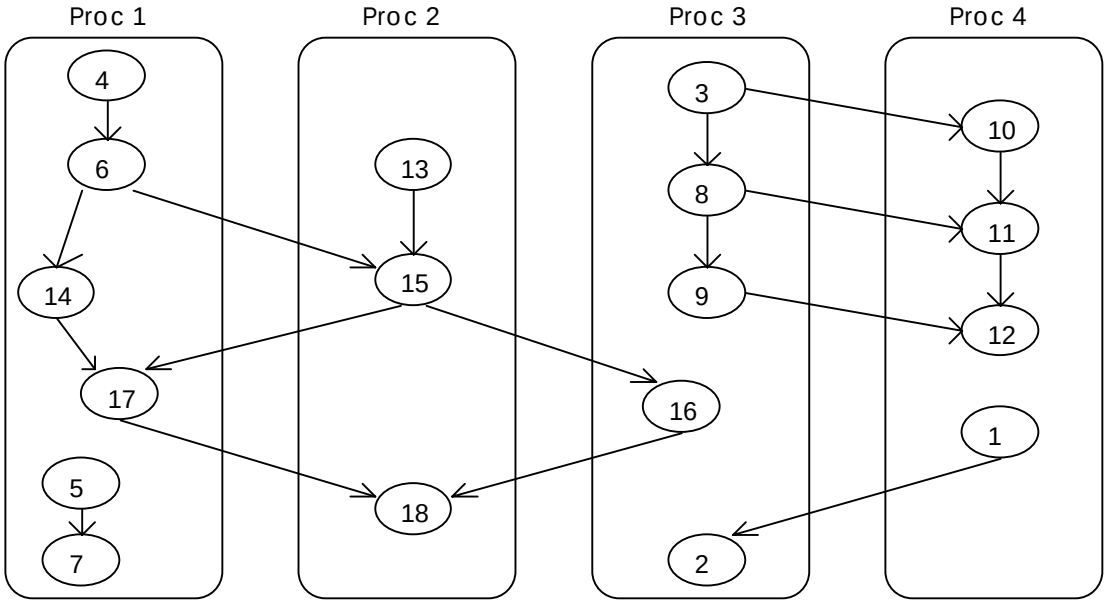

Figura 4.4 - Aplicação hipotética, onde Δ=7.

A tabela 4.5 mostra o resultado do cálculo quando o método convencional é utilizado. As relações de precedência são transformadas em *jitter* de liberação e as tarefas passam a ser tratadas como independentes. A coluna J_i mostra como o valor foi obtido. Por exemplo, a tarefa T_2 possui um *jitter* de liberação máximo 9 obtido da soma de 2 (tempo máximo de resposta de T_1) com 7 (tempo máximo para envio de uma mensagem entre diferentes processadores). Já a tarefa T_6 possui um *jitter* de liberação máximo 4 obtido da soma de 4 (tempo máximo de resposta de T_4) com zero (tempo máximo para envio de uma mensagem no mesmo processador). Quando uma tarefa possui mais de um predecessor direto, o pior cenário é usado. Por exemplo, T_{17} possui um *jitter* de liberação máximo devido a T_{14} de 29+0=29 e um *jitter* de liberação máximo devido a T_{15} de 27+7=34. O valor 34 é usado.

Tarefa	Atividade	Proc.	J_i	C_i	P_i	D_i
T_1	A_1	4	1	1	10	10
T_2	A_1	3	0	1	10	10
T_3	A_2	3	2	2	30	12
T_4	A_3	1	3	1	67	12
T_5	A_4	1	1	5	50	12
T_6	A_3	1	0	2	67	12
T_7	A_4	1	0	7	50	30
T_8	A_2	3	0	3	30	30
T_9	A_2	3	0	2	30	30
T_{10}	A_2	4	0	1	30	30
T_{11}	A_2	4	0	5	30	30
T_{12}	A_2	4	0	3	30	30
T_{13}	A_3	2	4	2	67	50
T_{14}	A_3	1	0	2	67	50
T_{15}	A_3	2	0	6	67	50
T_{16}	A_3	3	0	2	67	50
T_{17}	A_3	1	0	3	67	50
T_{18}	A_3	2	0	10	67	67

Tabela 4.4 - Aplicação hipotética.

Tarefa	J_i	C_i	P_i	D_i	R_i
T_1	1	1	10	10	1+1+0=2
T_2	2+7=9	1	10	10	9+1+0=10
T_3	2	2	30	12	2+2+2=6
T_4	3	1	67	12	3+1+0=4
T_5	1	5	50	12	1+5+1=7
T_6	4+0=4	2	67	12	4+2+6=12
T_7	7+0=7	7	50	30	7+7+8=22
T_8	6+0=6	3	30	30	6+3+4=13
T_9	13+0=13	2	30	30	13+2+7=22
T_{10}	6+7=13	1	30	30	13+1+1=15
T_{11}	13+7=20	5	30	30	20+5+2=27
T_{12}	22+7=29	3	30	30	29+3+8=40
T_{13}	4	2	67	50	4+2+0=6
T_{14}	12+0=12	2	67	50	12+2+15=29
T_{15}	12+7=19	6	67	50	19+6+2=27
T_{16}	27+7=34	2	67	50	34+2+9=45
T_{17}	27+7=34	3	67	50	34+3+17=54
T_{18}	54+7=61	10	67	67	61+10+8=79

Tabela 4.5 - Cálculo convencional do tempo máximo de resposta.

Ainda considerando a tabela 4.5, a coluna R_i mostra o tempo máximo de resposta formado por 3 componentes: o *jitter* de liberação máximo da tarefa, seu tempo de execução máximo e a interferência máxima que ela pode sofrer. Por exemplo, a tarefa T_5 possui um *jitter* de liberação máximo de 1, um tempo máximo de execução de 5 e pode receber, no pior caso, uma interferência da tarefa T_4 de 1. O resultado é um tempo máximo de resposta dado por 1+5+1=7.

A tabela 4.6 mostra os tempos máximo de resposta calculados através do algoritmo mostrado na seção anterior. A coluna J_i mostra o *jitter* de liberação máximo da tarefa equivalente, após a aplicação das transformações. Por exemplo, T_6 foi aglutinada com T_4 e herdou seu *jitter* de liberação máximo de 3. Já T_2 possui um *jitter* de liberação máximo devido ao tempo máximo de resposta de T_1 mais o tempo de envio da mensagem, ou seja, 2+7=9. Da mesma forma, a coluna C_i mostra o tempo máximo de execução da tarefa equivalente final. Por exemplo, T_9 foi aglutinada com T_3 e T_8, ficando a tarefa equivalente com um tempo máximo de execução dado por 2+3+2=7.

Ainda considerando a tabela 4.6, a coluna R_i também mostra o tempo máximo de resposta formado por 3 componentes: o *jitter* de liberação máximo da tarefa, seu tempo de execução máximo e a interferência máxima que ela pode sofrer. Por exemplo, a tarefa T_{12} possui um *jitter* de liberação máximo de 16 (devido a T_8), um tempo máximo de execução de 8 (devido a aglutinação de T_{11} e T_{12}) e pode receber, no pior caso, uma interferência da tarefa T_1 de 1. O resultado é um tempo máximo de resposta dado por 16+8+1=25.

A tabela 4.6 mostra que todas as tarefas da aplicação vão atender o respectivo deadline, ou seja, o seu tempo máximo de resposta é menor que o seu deadline. Entretanto, se o cálculo é feito da forma simplificada, a conclusão seria que as tarefas T_{12}, T_{17} e T_{18} poderiam perder seus deadlines. Uma aplicação considerada inviável pelo método simplificado é mostrada escalonável pelo método proposto neste trabalho.

Tarefa	J_i	C_i	P_i	D_i	R_i
T_1	1	1	10	10	1+1+0=2
T_2	2+7=9	1	10	10	9+1+0=10
T_3	2	2	30	12	2+2+2=6
T_4	3	1	67	12	3+1+0=4
T_5	1	5	50	12	1+5+1=7
T_6	3	1+2=3	67	12	3+3+5=11
T_7	1	5+7=12	50	30	1+12+3=16
T_8	2	2+3=5	30	30	2+5+2=9
T_9	2	2+3+2=7	30	30	2+7+2=11
T_{10}	6+7=13	1	30	30	13+1+1=15
T_{11}	9+7=16	5	30	30	16+5+1=22
T_{12}	9+7=16	5+3=8	30	30	16+8+1=25
T_{13}	4	2	67	50	4+2+0=6
T_{14}	3	1+2+2=5	67	50	3+5+12=20
T_{15}	11+7=18	6	67	50	18+6+0=24
T_{16}	24+7=31	2	67	50	31+2+9=42
T_{17}	24+7=31	3	67	50	31+3+12=46
T_{18}	46+7=53	10	67	67	53+10+0=63

Tabela 4.6 - Cálculo proposto neste trabalho do tempo máximo de resposta.

Mais importante é comparar diretamente os tempos máximo de resposta calculados pelos dois métodos. A tabela 4.7 coloca estes valores lado a lado. É possível perceber que o método proposto gera sempre tempos máximo de resposta menores ou iguais ao método convencional. A medida que são consideradas tarefas posicionadas mais ao fim das atividades, a vantagem de usar o método proposto aumenta.

Tarefa	R_i Método Convencional	R_i Método Proposto
T_1	2	2
T_2	10	10
T_3	6	6
T_4	4	4
T_5	7	7
T_6	12	11
T_7	22	16
T_8	13	9
T_9	22	11
T_{10}	15	15
T_{11}	27	22
T_{12}	40	25
T_{13}	6	6
T_{14}	29	20
T_{15}	27	24
T_{16}	45	42
T_{17}	54	46
T_{18}	79	63

Tabela 4.7 - Comparação dos cálculos convencional e proposto.

4.8 Conclusões

Este capítulo apresentou um método para o cálculo do tempo máximo de resposta de tarefas imprecisas em ambiente distribuído, quando apenas as partes obrigatórias são consideradas. É suposto que tais tarefas possuem relações de precedência arbitrárias, liberação dinâmica e são escalonadas segundo o mecanismo de prioridades fixa. São consideradas tarefas periódicas e esporádicas, com deadline menor ou igual ao período.

Diversos teoremas foram demonstrados com o propósito de provar a corretude do método. Cada teorema descreve um tipo de transformação que pode ser empregado sobre a aplicação em questão. Ao final de uma série de transformações, a aplicação original terá sido transformada em uma aplicação equivalente formada apenas por tarefas independentes. Neste momento, soluções encontradas na literatura para a análise da escalonabilidade de tarefas independentes em monoprocessadores podem ser aplicadas. A sequência na qual as transformações devem ser realizadas é definida pelo algoritmo apresentado na seção 4.6.

Um exemplo composto por 18 tarefas executadas de forma distribuída em 4 processadores foi utilizado para ilustrar a aplicação do algoritmo proposto e sua vantagem sobre a aplicação de técnicas mais simples. O algorítimo introduzido neste trabalho é menos pessimista que uma aplicação simplista de técnicas descritas na literatura. Sua aplicação resultou, no exemplo, em tempos máximos de resposta mais próximos do valor real (menos pessimistas).

É importante notar que o algoritmo desenvolvido neste capítulo pode ser também empregado na análise da escalonabilidade de tarefas normais, isto é, tarefas compostas somente por partes obrigatórias. Neste caso, a solução para o problema do escalonamento como um todo se resume às etapas de alocação e garantia em tempo de projeto.

5 Teste de Aceitação para Partes Opcionais

5.1 Introdução

O emprego da Computação Imprecisa como técnica de escalonamento exige que, em tempo de execução, a capacidade ociosa ("spare capacity") dos processadores seja detectada. Esta capacidade ociosa será então utilizada pela política de admissão na execução de partes opcionais. O componente responsável pela identificação da capacidade ociosa é o teste de aceitação. É o teste de aceitação que determina se uma dada parte opcional pode ser executada sem comprometer as garantias fornecidas para partes obrigatórias e para partes opcionais previamente aceitas.

O teste de aceitação é usado em tempo de execução. Logo, seu custo computacional ("overhead") deve ser baixo. De outra forma, a capacidade ociosa dos processadores será consumida pelo próprio teste de aceitação, ao invés de ser usada pelas partes opcionais. Uma solução ótima para o teste de aceitação é definida como aquela capaz de detectar toda a capacidade ociosa existente tão logo ela seja gerada. Como será visto na próxima seção, soluções ótimas para o teste de aceitação são inviáveis na prática, em função da complexidade de seus algoritmos. Existe na literatura um conjunto de soluções de compromisso para o problema.

O objetivo deste capítulo é apresentar uma solução para o teste de aceitação que seja compatível com o modelo de tarefas adotado. Ao mesmo tempo, a solução apresentada deverá oferecer um compromisso razoável entre o custo computacional do algoritmo ("overhead") e a quantidade de capacidade ociosa que ele consegue realmente detectar.

A seção 5.2 faz uma revisão da literatura relacionada com o assunto. A seção 5.3 descreve a abordagem a ser adotada. A seção 5.4 analisa as implicações sobre a abordagem escolhida das características do modelo de tarefas adotado. Principalmente, os aspectos distribuição e relações de precedência. Um exemplo numérico é apresentado na seção 5.5. A seção 5.6 discute a solução proposta neste capítulo para o teste de aceitação. Os comentários finais aparecem na seção 5.7.

5.2 Revisão da Literatura

Existem na literatura diversas propostas com respeito à identificação de capacidade ociosa em tempo de execução. Nesta seção estas propostas são identificadas e analisadas sob a perspectiva de seu possível emprego em um contexto de Computação Imprecisa. Em particular, seu possível emprego no contexto definido pelo modelo de tarefas adotado neste trabalho.

Muitos trabalhos na literatura procuram garantir, em tempo de execução, tarefas aperiódicas. Embora tais algoritmos não tenham sido criados especificamente para Computação Imprecisa, eles podem ser utilizados como testes de aceitação para partes opcionais. Isto é possível pois, do ponto de vista do escalonador, uma parte opcional pode ser considerada como uma tarefa aperiódica que não foi garantida em tempo de projeto e que necessita de uma garantia dinâmica. Ao invés de utilizar o recurso garantido

dinamicamente para executar uma nova tarefa, ele é usado para aumentar o tempo de execução de uma tarefa existente.

Soluções Baseadas em Planejamento

Em [ZHA 87] e [RAM 90] são apresentados algoritmos capazes de oferecer uma garantia dinâmica para tarefas aperiódicas a partir da manipulação explícita da escala de execução. É suposta uma aplicação composta por um conjunto de tarefas periódicas garantidas em tempo de projeto mais um conjunto de tarefas aperiódicas que devem receber garantia dinâmica. Estes algoritmos não usam prioridades. Eles mantém uma tabela onde está anotado quando cada tarefa vai ocupar o processador. Ao contrário da grade utilizada em executivos cíclicos (seção 2.3), esta tabela é construida dinamicamente, em tempo de execução. Ela descreve como o processador será ocupado pelas próximas **M** unidades de tempo, onde **M** é o mínimo múltiplo comum dos períodos das tarefas periódicas.

Quando chega uma tarefa aperiódica, a tabela é manipulada no sentido de localizar uma lacuna suficientemente grande para encaixar a tarefa aperiódica que chega. A manipulação da tabela pode incluir o deslocamento das tarefas dentro do plano de execução, desde que seus deadlines não sejam comprometidos. Caso o algoritmo tenha sucesso em identificar uma capacidade ociosa suficiente, a tarefa aperiódica é aceita. O momento de sua futura execução é então assinalado na própria tabela.

Existem variações na forma como a tabela é organizada e na forma como ela é pesquisada em busca de capacidade ociosa. Muitas vezes este tipo de algoritmo é chamado de baseado em planejamento ("planning based") exatamente por "planejar" a futura utilização do processador. Algoritmos deste tipo não são ótimos, no sentido que tarefas aperiódicas que poderiam ser aceitas são, as vezes, rejeitadas. Isto é necessário pois um algoritmo ótimo é inviável em tempo de execução devido ao seu custo computacional.

Soluções baseadas em planejamento não são satisfatórias para o modelo de tarefas adotado neste trabalho por diversas razões. Tarefas esporádicas garantidas em tempo de projeto são problemáticas pois não é possível conhecer o instante de sua liberação antes que ele aconteça. Desta forma, não é possível "planejar" antecipadamente o momento de sua execução. O fato de tarefas possuirem elevado *jitter* de liberação também causa o mesmo problema. O fato da estrutura de dados mantida ser proporcional ao mínimo múltiplo comum dos períodos é problemático quando os períodos das tarefas são números primos entre si. Finalmente, em sistemas baseados em planejamento é a escala de execução planejada que decide quem executa quando. Considerando o modelo de tarefas adotado neste trabalho, seria necessário manter uma coerência entre as prioridades das tarefas e o plano construido, o que aumenta a complexidade dos algoritmos utilizados em tempo de execução.

Soluções Baseadas em Servidores

Em [LEH 87] e [SPR 89] são apresentados servidores capazes de oferecer garantia dinâmica para tarefas aperiódicas no contexto de escalonamento baseado em prioridades fixas. Após todas as tarefas periódicas terem sido garantidas, ainda em tempo de projeto estes servidores reservam para si a capacidade restante no sistema. Em tempo de execução, estes servidores utilizam esta capacidade restante para executar tarefas aperiódicas. Um

teste de aceitação é construido com base nos recursos que o servidor dispõe a cada momento.

Os servidores falham em capturar outras fontes de recurso do sistema, tais como o tempo ganho na execução ("gain time"). Em situações práticas os processadores são dimensionados de forma que, quando consideramos o comportamento em pior caso das partes obrigatórias, a capacidade restante é pequena. Nestes casos, espera-se executar as partes opcionais com os recursos liberados na medida que as partes obrigatórias não ocupam todo o recurso reservado, ou seja, apresentam um comportamento melhor que o pior caso. Isto é possível pois o tempo de execução no pior caso de uma tarefa é, em geral, várias vezes superior ao seu tempo médio de execução. Por exemplo, não é absurdo projetar um sistema onde as partes obrigatórias ocupam, no pior caso, 90% da capacidade do sistema e ter-se, em tempo de execução, apenas 50% realmente utilizado. Neste caso, um servidor teria uma capacidade, definida em tempo de projeto, equivalente à apenas 10% da capacidade do sistema. Além do tempo ganho na execução, as diferenças de fase entre as tarefas geram uma capacidade ociosa temporária que também não é capturada pelos servidores ([BUR 96]).

Tomada Estática de Folga

Em [LEH 92], [THU 93] e [THU 94] é apresentada a tomada estática de folga ("static slack stealing") como um esquema capaz de identificar capacidade ociosa no sistema no contexto de prioridades fixas. Esta capacidade ociosa pode então ser usada para escalonar tarefas aperiódicas. É suposta uma aplicação composta por um conjunto de tarefas periódicas garantidas em tempo de projeto mais um conjunto de tarefas aperiódicas que devem receber garantia dinâmica. Em tempo de projeto é construida uma tabela de folgas que indica em que momentos o sistema admite a execução de tarefas aperiódicas sem comprometer as tarefas previamente garantidas. Na medida que as tarefas são executadas, esta tabela é mantida atualizada. Desta forma, quando uma tarefa apresenta um comportamento melhor que o pior caso, as folgas descritas pela tabela são ampliadas. De forma semelhante, a aceitação de uma tarefa aperiódica implica no consumo de folga e, portanto, na atualização da tabela. Uma variação do algoritmo básico é apresentada em [TIA 94].

A tomada estática de folga é capaz de gerar um teste de aceitação com excelente eficiência no que diz respeito à identificação de folgas. Entretanto, a aplicação de tomada estática de folga está restrita a conjuntos de tarefas estritamente periódicas. Além disto, o tamanho da tabela de folgas construida em tempo de projeto e mantida durante a execução é proporcional ao mínimo múltiplo comum dos períodos das tarefas. Esta tabela poderá possuir um tamanho exageradamente grande quando os períodos forem números primos entre si.

Tomada Dinâmica de Folga

Em [DAV 93a] é apresentada a tomada dinâmica de folga ("dynamic slack stealing"). Naquele trabalho, o princípio da tomada estática de folga é adaptado para um modelo de tarefas que inclui sincronização entre tarefas, *jitter* na liberação e tarefas esporádicas garantidas. É incluída na folga todo o tempo ganho quando uma tarefa não usa todo o tempo de processador que havia sido reservado para ela ("gain time"). Esta ampliação do modelo de tarefas suportado, em relação à tomada estática de folga, foi

possível na medida que o algoritmo proposto calcula as folgas em tempo de execução. A tabela mantida em tempo de execução é proporcional ao número de tarefas e não ao MMC de seus períodos. O algoritmo de tomada dinâmica de folga também resulta em excelente eficiência com relação à identificação das folgas no sistema. Entretanto, seu custo computacional ("overhead") é muito elevado, o que inviabiliza sua utilização na prática.

Tomada Dinâmica de Folga Aproximada

Em [DAV 93b] e [AUD 94b] é proposta uma solução aproximada para o algoritmo de tomada dinâmica de folga. Esta aproximação é tal que, toda capacidade ociosa identificada realmente existe. Entretanto, este algoritmo apresenta uma eficiência inferior ao algoritmo anterior, pois ele não é capaz de identificar todas as folgas existentes no sistema em um dado momento. As simulações apresentadas em [DAV 93b] mostram que, mesmo não sendo uma solução ótima, a tomada dinâmica de folga aproximada resulta em eficiência superior às soluções baseadas em servidores, no que diz respeito à aceitação de tarefas aperiódicas. Como no caso anterior, o modelo de tarefas suportado inclui tarefas esporádicas e *jitter* na liberação. Entretanto, o custo computacional da solução aproximada é muito menor do que o custo da tomada dinâmica de folga.

5.3 Descrição da Abordagem

Neste trabalho será empregada a técnica de tomada dinâmica de folga aproximada (DASS, "dynamic approximate slack stealing"), proposta em [DAV 93b]. Através da discussão apresentada na seção anterior é possível observar que o DASS é um compromisso entre eficiência na identificação de folgas e custo computacional. Ele resolve de forma satisfatória a questão de como determinar que a execução de uma parte opcional não irá comprometer a execução das partes obrigatórias e opcionais previamente garantidas. Além disto, o modelo de tarefas suportado pelo DASS é o mais próximo do modelo de tarefas adotado neste trabalho.

No nosso modelo, sempre que uma tarefa vai iniciar sua execução ocorre uma solicitação automática de garantia para a sua parte opcional. Esta solicitação é feita para um tempo de execução O_i e deadline igual ao deadline D_i da parte obrigatória. O algoritmo DASS mantém contadores de folga $S_i(t)$ para cada nível **i** de prioridade. No momento de aceitar ou não a parte opcional da tarefa T_i, os contadores de folga associados com níveis de prioridades iguais ou inferiores ao nível **i** são consultados. Considerando que números maiores indicam prioridade inferior, a parte opcional de uma tarefa T_i poderá ser aceita quando $\forall j \geq i \; . \; O_i \leq S_j(t)$, onde **t** é o instante que o teste de aceitação é executado. A seção 5.3.1 descreve como os valores $S_i(t)$ são calculados pelo DASS.

5.3.1 Cáculo das Folgas pelo DASS

Esta seção descreve como o DASS calcula e mantém os contadores de folga para cada nível de prioridade. Em tempo de execução o suporte de execução mantém para cada tarefa os seguintes valores:

$x_i(t)$: Instante, em valores absolutos, da próxima liberação de T_i, supondo que ela ocorra o mais cedo possível;

$d_i(t)$: Instante, em valores absolutos, do próximo deadline de T_i a ser cumprido;

$c_i(t)$: Tempo máximo de execução restante na atual invocação de T_i.

A partir destes valores é possível calcular a folga existente no nível **i** em um dado instante **t**, denotada por $S_i(t)$. Este valor representa quanta interferência extra T_i pode receber e ainda cumprir o próximo deadline $d_i(t)$. Como o algoritmo de cálculo usado é aproximado, $S_i(t)$ é na verdade um limite mínimo ("lower bound") para este valor.

O valor de $S_i(t)$ é dado por:

$$S_i(t)=\left[d_i(t)-t-\sum_{\forall j\le i} I_j\left(t,d_i(t)\right)\right]_0$$

onde $I_j(t,d_i(t))$ representa o tempo máximo de execução da tarefa T_j dentro do intervalo $[t,t+d_i(t))$. A notação $[\ a\ ]_0$ é definida como:

a quando $a \ge 0$;
0 quando $a < 0$.

O valor $I_j(t,d_i(t))$ é dado por:

$$I_j\left(t,d_i(t)\right)= c_j(t)+f_j\left(t,d_i(t)\right)\times C_j + Min\left(C_j,\left(d_i(t)-x_j(t)-f_j(t,d_i(t))\times P_j\right)_0\right)$$

onde $f_j(t,d_i(t))$ é o número de invocações completas de T_j dentro do intervalo $[t,d_i(t))$, dado por:

$$f_j\left(t,d_i(t)\right)=\left\lfloor \left(d_i(t)-x_j(t)\right)/P_j\right\rfloor_0 .$$

O valor $S_i(t)$ é calculado sempre que a invocação corrente de T_i é concluida. Neste momento, qualquer tarefa com prioridade superior a $\rho(T_i)$ já terá concluido e estará esperando a próxima chegada ou liberação. Caso uma tarefa com prioridade superior estivesse liberada, seria ela executando no lugar de T_i, e T_i não estaria concluindo. Desta forma, temos que no cálculo o valor $c_j(t)$ será sempre zero (T_j aguarda nova chegada) ou C_j (T_j já chegou mas aguarda a liberação).

Entre um cálculo e outro, o valor das folgas de cada tarefa deve ser mantido atualizado, em função das folgas que são geradas ou consumidas. A cada chaveamento de contexto a tabela de folgas será atualizada. Suponha que ocorre um chaveamento de contexto após a tarefa T_i executar por **E** unidades de tempo. Neste caso, a folga de todas as tarefas com prioridade superior à T_i deve ser diminuida de **E** unidades. Em outras palavras, $\forall j<i$, $S_j(t) = S_j(t) - E$. Suponha ainda que a tarefa T_i executou **F** unidades de tempo a menos que o seu tempo de execução no pior caso. Neste caso, a folga de todas as tarefas com prioridade igual ou inferior à T_i deve ser aumentada de **F** unidades: $\forall j\ge i$, $S_j(t)=S_j(t)+F$.

Uma parte opcional com prioridade **i** poderá ser garantida se a folga existente no nível **i** for maior que o seu tempo de execução no pior caso. Além disto, caso seja aceita, esta parte opcional vai gerar interferência sobre os níveis de **i+1** até **n**. Logo, é necessário que a folga existente nestes níveis também seja maior ou igual ao tempo de computação solicitado. Em outras palavras, uma parte opcional O_i poderá ser garantida quando:

$$O_i \le \underset{\forall j\ge i}{Min} S_j(t) .$$

Caso a parte opcional O_i seja aceita, é necessário corrigir a tabela de folgas fazendo:

$$\forall j \geq i, S_j(t) = S_j(t) - O_i \ .$$

Observe que a garantia para a parte opcional de uma liberação $T_{i,k}$ é dada no início da execução da tarefa e não no instante da liberação. Isto é feito para aproveitar, no teste de aceitação, das folgas ("slack") geradas pelas tarefas de maior prioridade que executam e são concluidas exatamente entre a liberação e o início de $T_{i,k}$. Esta garantia dada no início da execução da tarefa não pode ser revogada, dentro do modelo das múltiplas versões.

O modelo de tarefas adotado neste trabalho difere em parte do modelo de tarefas descrito em [DAV 93b] e usado para definir o DASS. Em particular, no modelo de tarefas usado em [DAV 93b] não existem relações de precedência e a aplicação executa em um único processador. Ambos os modelos possuem em comum o fato de trabalharem com prioridades fixas, tarefas periódicas e esporádicas, que podem apresentar *jitter* na liberação. A próxima seção mostra como o algoritmo DASS pode ser adaptado para o modelo de tarefas empregado neste trabalho.

5.4 Adaptação para o Modelo de Tarefas Adotado

O algoritmo DASS foi concebido para aplicações que não incluem relações de precedência e são executadas em ambiente monoprocessado. Logo, não é possível aplicar diretamente o DASS ao modelo de tarefas adotado neste trabalho. Nesta seção são discutidas as mudanças necessárias no DASS para que ele possa ser aplicado ao contexto deste trabalho.

Relações de Precedência

Basicamente, o algoritmo DASS calcula qual a quantidade de interferência extra que cada tarefa pode receber sem aumentar o tempo máximo de resposta além do seu deadline. Isto significa que, ao aplicar o DASS original, o tempo máximo de resposta das tarefas poderá aumentar, mas nunca passar do respectivo deadline.

Na análise de escalonabilidade proposta no capítulo 4, a eliminação das relações de precedência entre diferentes processadores somente foi possível devido ao fato delas serem substituidas por *jitter* na liberação da tarefa sucessora. *Jitter* de liberação foi adicionado em uma quantidade igual ao tempo máximo de resposta de tarefa predecessora.

No momento em que o emprego do DASS amplia o tempo máximo de resposta de uma tarefa, por aceitar algumas partes opcionais, seria necessário ampliar também os *jitter* de liberação associados com estes tempos máximos de resposta e então recalcular a escalonabilidade de todo o sistema. Isto é inviável na prática pois o fenômeno ocorre em tempo de execução e de forma distribuída. A aceitação de uma parte opcional no processador **p** vai aumentar os tempos de resposta das tarefas que executam em **p**. Por sua vez, isto vai aumentar o *jitter* na liberação de suas sucessoras remotas, dificultando o escalonamento das tarefas em um outro processador qualquer.

Por exemplo, considere a aplicação hipotética descrita no capítulo anterior e reproduzida pela figura 5.1. Caso a execução da parte opcional de T_6 aumente o tempo

máximo de resposta de T_6, então seria aumentado o *jitter* na liberação de T_{15}. Com um *jitter* na liberação aumentado, o tempo máximo de resposta de T_{15} também seria maior. Em um efeito cascata, o mesmo acontece com T_{16}. É até possível que o novo tempo máximo de resposta de T_{15} ou T_{16} venha a ser maior que o respectivo deadline. Uma aplicação do DASS considera o efeito local da interferência extra gerada, mas não considera o efeito propagado através das relações de precedência. Através de relações de precedência inter-processadores teremos este efeito espalhado por vários processadores do sistema.

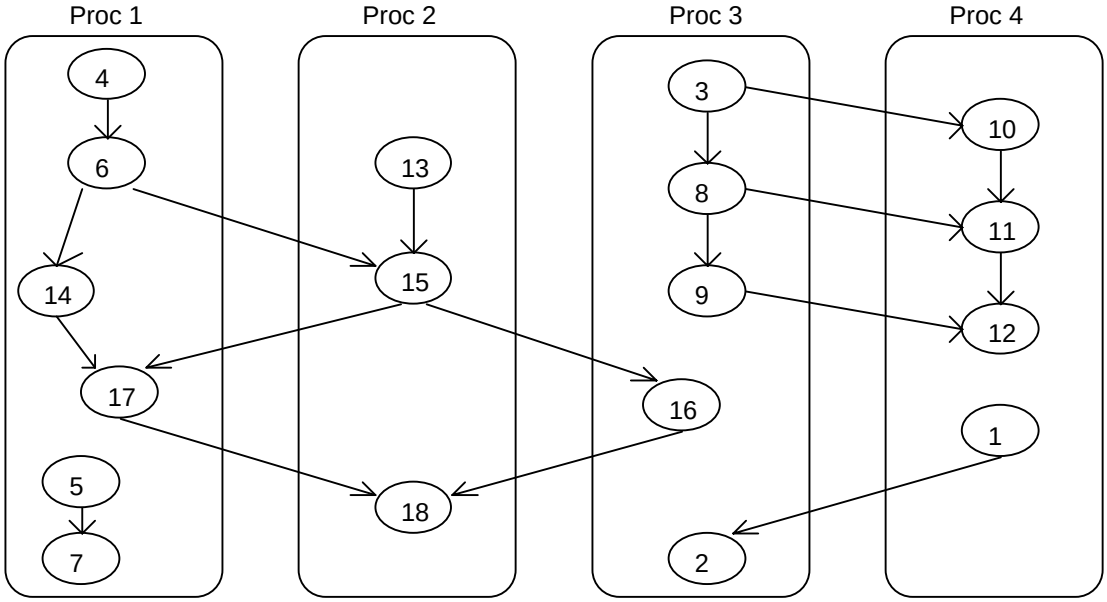

Figura 5.1 - Aplicação hipotética descrita no capítulo 4.

Este problema não ocorreria caso o *jitter* de liberação das tarefas sucessoras fosse definido em função do deadline da tarefa predecessora e não de seu tempo máximo de resposta. Como a aplicação do DASS não invalida os deadlines, o cálculo realizado em tempo de projeto continuaria válido. Entretanto, como o deadline de uma tarefa pode ser muito maior que o seu tempo máximo de resposta, estariamos aumentando o pessimismo da análise e possivelmente tornando um sistema escalonável em não escalonável.

Uma forma de contornar o problema é usar o tempo máximo de resposta no lugar do deadline nas equações que calculam a folga de uma tarefa que possui sucessores remotos. Em outras palavras, se $\exists T_j \in Suc(T_i).H(T_j) \neq H(T_i)$ então R_i será usado no lugar de D_i para o cálculo de $S_i(t)$. Esta informação (usar R_i ou D_i) depende apenas das propriedades estáticas da tarefa (suas relações de precedência) e pode ser anotada no descritor da tarefa, no momento que ela é criada.

Considerando ainda a figura 5.1, a tarefa T_6 somente poderá ser executada de forma precisa caso o seu tempo máximo de resposta não seja afetado. Com isto estaremos garantindo que a tarefa T_{15} receberá a mensagem de T_6, no pior caso, como previsto na análise em tempo de projeto. Como as tarefas com prioridade superior à T_6 (por exemplo T_5) somente executam suas partes opcionais caso T_6 tenha folga para tanto, temos que a execução precisa de outras tarefas também não conseguirá aumentar o tempo de resposta de T_6 além do máximo previsto em projeto.

Em resumo, no DASS modificado (MDASS), uma parte opcional somente será aceita caso a sua execução não altere o tempo máximo de resposta das tarefas locais com sucessores remotos e não viole o deadline das demais tarefas locais. A própria tarefa cujo tempo de execução está sendo estendido é incluida na análise. Este comportamento difere do DASS normal, que aceita partes opcionais caso os tempos máximos de resposta sejam alterados, mas não acima do respectivo deadline.

É possível capturar a capacidade restante, através de um aumento artificial, durante o teste de escalonabilidade, do tempo máximo de resposta das tarefas com sucessores remotos. Este aumento deve ser feito de maneira que o sistema como um todo permaneça escalonável. O tempo máximo de resposta artificialmente aumentado de uma tarefa vai permitir a inclusão, durante a execução desta tarefa, de interferência causada por partes opcionais dela própria ou de tarefas com prioridade superior. Esta alteração resultaria na melhora da capacidade do MDASS de identificar folgas no sistema.

Caso no instante **t** a parte opcional da tarefa T_i seja aceita pelo teste do MDASS, isto significa que é possível executar uma computação com tempo máximo de execução O_i na prioridade **i** sendo garantido que:

- A computação O_i será concluida antes do seu deadline D_i;
- Nenhum tempo máximo de resposta de tarefa com sucessor remoto será alterado e, por consequência, nenhum deadline será perdido em outros processadores.

Tarefas Esporádicas

A literatura que descreve o DASS também não discute as alterações necessárias para quando a aplicação inclui tarefas esporádicas. Neste caso, o único cuidado adicional se refere ao cálculo do momento futuro no qual deverá ocorrer a próxima liberação da tarefa esporádica. Neste aspecto, a diferença básica entre tarefas periódicas e tarefas esporádicas é que as tarefas periódicas possuem todos os seus momentos de chegada previamente fixados, ainda em tempo de projeto. Por exemplo, suponha que T_i é uma tarefa periódica com período P_i. Em qualquer instante **t** é possível determinar com certeza o próximo instante de chegada de T_i. A sua liberação ocorrerá antes caso seu *jitter* de liberação seja zero. Logo, $x_i(t)$ corresponde exatamente ao instante da próxima chegada de T_i, isto é, $\lceil t / P_i \rceil \times P_i$.

Suponha agora que T_j é uma tarefa esporádica. Neste caso, $x_j(t)$ deve indicar este momento de liberação futura sem que os instantes de chegada sejam conhecidos. Apenas os instantes de liberação passados são conhecidos. Suponha que, no instante **t**, o valor **y** representa o momento da última liberação de T_j. Obviamente, **y** é conhecido do escalonador. No pior caso, a liberação que ocorreu em **y** apresentou um *jitter* de liberação máximo. Isto vai permitir que a tarefa T_j sejá liberada novamente em $x_j(t) = y + P_j - J_j$.

Dependência Inter-Tarefa

A dependência inter-tarefa, descrita no capítulo 3, pode ser usada para aumentar a eficiência do MDASS na identificação de folgas. Suponha que exista uma relação de precedência entre T_j e T_i de tal sorte que uma execução precisa de T_j reduz o tempo de execução de T_i. Neste caso, o modelo de tarefas adotado supõe a existência dos fatores de redução $\beta_{j,i}$ e $\gamma_{j,i}$ ligando T_j à T_i.

No momento em que a política de admissão e o teste de aceitação determinam que a k-ésima ativação da tarefa T_j será executada precisamente, já estará sendo determinada também uma redução no tempo máximo de execução da tarefa T_i. A parte obrigatória da k-ésima ativação de T_i terá uma redução determinada pelo fator $\beta_{j,i}$. Esta economia gerada seria normalmente transformada em folga ao término da tarefa T_i, quando o tempo ganho é identificado e a tabela de folgas atualizada. Entretanto, esta economia já é conhecida no momento que T_j inicia sua execução. Caso T_i seja executada no mesmo processador que T_j, quando T_j inicia podemos imediatamente atualizar a tabela de folgas e contabilizar a redução gerada em T_i pela execução precisa de T_j. Caso T_j e T_i executem em processadores diferentes, a folga é causada por uma execução precisa de T_j em $H(T_j)$, mas ela vai acontecer realmente em $H(T_i)$. Neste caso, a folga somente será conhecida quando a mensagem enviada de T_j para T_i, ao término de sua execução, chegar em $H(T_i)$.

A redução da parte opcional de T_i, em função do fator $\gamma_{j,i}$, também pode ser aproveitada. Neste caso, quando T_i vai iniciar sua execução, a decisão de executar ou não a sua parte opcional já considera o fato da duração da parte opcional ter sido reduzida por uma execução precisa de T_j.

5.5 Exemplo Numérico

Vamos ilustrar o emprego do MDASS através de um exemplo numérico. Considere a aplicação hipotética reproduzida pela figura 5.1. Suponha que no instante zero de tempo as tarefas T_4 e T_5 são liberadas. Suponha ainda que T_4 executa durante uma unidade de tempo (seu pior caso) e a tarefa T_5 executa durante 4 unidades de tempo (uma a menos que o pior caso). A figura 5.2 mostra a linha de tempo até o instante 5, quando T_5 é concluida. As características temporais das tarefas aparecem na tabela 5.1.

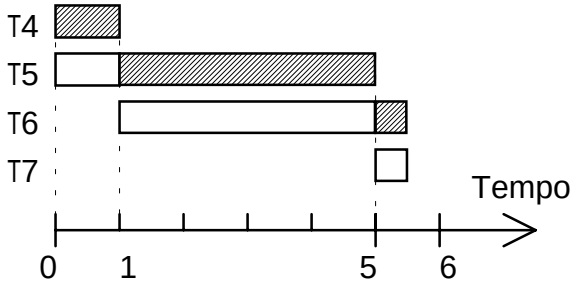

Figura 5.2 - Linha de tempo possível para a aplicação hipotética.

No instante 5 a tarefa T_6 vai iniciar sua execução. Vamos supor que neste momento a política de admissão resolve que é importante para o sistema executar T_6 de maneira precisa. Suponha ainda que O_6=1. A questão básica é se a ampliação de T_6 em 1 unidade de tempo vai comprometer ou não a escalonabilidade da aplicação. As tarefas T_4, T_6, T_{14} e T_{17} possuem sucessores remotos e, portanto, precisam ter o seu o tempo máximo de resposta preservado. As tarefas T_5 e T_7 não possuem sucessor remoto e, portanto, precisam apenas ter o seu deadline preservado.

Vamos reproduzir o comportamento do MDASS do instante zero até o instante 5, momento no qual é solicitado recurso para a parte opcional da tarefa T_6. No instante zero o sistema inicia supondo folga zero para todas as tarefas. Isto é:
$S_4(0)=0$, $S_5(0)=0$, $S_6(0)=0$, $S_7(0)=0$, $S_{14}(0)=0$, $S_{17}(0)=0$.

Tarefa	J_i	C_i	P_i	D_i	R_i
T_1	1	1	10	10	1+1+0=2
T_2	2+7=9	1	10	10	9+1+0=10
T_3	2	2	30	12	2+2+2=6
T_4	3	1	67	12	3+1+0=4
T_5	1	5	50	12	1+5+1=7
T_6	3	1+2=3	67	12	3+3+5=11
T_7	1	5+7=12	50	30	1+12+3=16
T_8	2	2+3=5	30	30	2+5+2=9
T_9	2	2+3+2=7	30	30	2+7+2=11
T_{10}	6+7=13	1	30	30	13+1+1=15
T_{11}	9+7=16	5	30	30	16+5+1=22
T_{12}	9+7=16	5+3=8	30	30	16+8+1=25
T_{13}	4	2	67	50	4+2+0=6
T_{14}	3	1+2+2=5	67	50	3+5+12=20
T_{15}	11+7=18	6	67	50	18+6+0=24
T_{16}	24+7=31	2	67	50	31+2+9=42
T_{17}	24+7=31	3	67	50	31+3+12=46
T_{18}	46+7=53	10	67	67	53+10+0=63

Tabela 5.1 - Características da aplicação hipotética.

No instante 1 a tarefa T_4 termina e é necessário atualizar a tabela de folgas. Todas as tarefas com prioridade maior que $\rho(T_4)$ devem ter sua folga decrementada de E=1, ou seja, o tempo que T_4 ocupou o processador. Também todas as tarefas com prioridade menor que $\rho(T_4)$ devem ter sua folga incrementada de F=0, ou seja, o tempo reservado por T_4 mas não utilizado. Estas duas atualizações na verdade não alteram a tabela de folgas.

Finalmente, é necessário recalcular a folga da própria tarefa T_4. São utilizadas as equações da seção 5.3, com exceção do valor $d_i(t)$. Como descrito na seção anterior, $d_i(t)$ passa a representar o instante do próximo tempo máximo de resposta de T_i a ser cumprido, quando T_i possui sucessores remotos. Supondo que a atividade que inclui T_4 é periódica, no instante 1 temos:

$x_4(1) = 67, \quad d_4(1) = 67+4 = 71, \quad c_4(1) = 0.$

Estes números devem ser interpretados da seguinte forma:

- Supondo que a tarefa T_4 seja periódica, sua próxima chegada ocorrerá em 67. Supondo *jitter* de liberação zero, teremos a sua próxima liberação, no pior caso, em 67.
- O próximo tempo máximo de resposta que esta tarefa deverá cumprir é em 71, após sua próxima chegada, pois o tempo máximo de resposta atual já foi cumprido (a tarefa foi concluida).
- Do momento atual até o instante de sua nova chegada, esta tarefa representa uma interferência zero para as demais tarefas do processador.

A partir destes valores podemos calcular a folga da tarefa T_4 no instante 1. Observe que nenhuma outra tarefa interfere com T_4.

$$S_4(1)=\left[d_4(1)-t-\sum_{\forall j\le 4} I_j\left(1,d_4(1)\right)\right]_0$$

$$S_4(1)=\left[71-1-I_4\left(1,d_4(1)\right)\right]_0$$
$$I_4\left(1,d_4(1)\right)=c_4(1)+f_4\left(1,d_4(1)\right)\times 1+\mathrm{Min}\left(1,\left(d_4(1)-x_4(1)-f_4(1,d_4(1))\times P_4\right)_0\right)$$
$$I_4\left(1,71\right)=0+f_4\left(1,71\right)\times 1+\mathrm{Min}\left(1,\left(71-67-f_4(1,71)\times 67\right)_0\right)$$
$$f_4\left(1,71\right)=\left\lfloor\left(71-67\right)\div 67\right\rfloor_0=0$$
$$I_4\left(1,71\right)=0+0\times 1+\mathrm{Min}\left(1,\left(71-67-0\right)_0\right)=0+0+1=1$$
$$S_4(1)=\left[71-1-1\right]_0=69$$

O valor $S_4(1)=69$ deve ser interpretado da seguinte forma: se do instante 1 até o instante 71 a tarefa T_4 receber uma interferência extra de 69 unidades de tempo, ainda assim ela será capaz de cumprir seu próximo tempo máximo de resposta, no instante 71. Por interferência extra entende-se interferência além daquela suposta durante a analise de escalonabilidade realizada em tempo de projeto. Após as atualizações em função da conclusão da tarefa T_4, temos:

$S_4(1)=69$, $S_5(1)=0$, $S_6(1)=0$, $S_7(1)=0$, $S_{14}(1)=0$, $S_{17}(1)=0$.

No instante 5 a tarefa T_5 termina e é novamente necessário atualizar a tabela de folgas. Todas as tarefas com prioridade maior que $\rho(T_5)$ devem ter sua folga decrementada de E=4, ou seja, o tempo que T_5 ocupou o processador. Também todas as tarefas com prioridade menor ou igual a $\rho(T_5)$ devem ter sua folga incrementada de F=1, ou seja, o tempo reservado por T_5 mas não utilizado. Estas duas atualizações alteram a tabela de folgas da seguinte forma:

$S_4(5)=65$, $S_5(5)=1$, $S_6(5)=1$, $S_7(5)=1$, $S_{14}(5)=1$, $S_{17}(5)=1$.

Também é necessário recalcular a folga da própria tarefa T_5. A atualização feita em $S_5(5)$ no passo anterior, neste caso, foi inútil. Entretanto, ela é necessária quando o chaveamento de contexto ocorre antes da tarefa em execução estar concluida. Existem propostas na literatura ([AUD 94a]) que advogam a inclusão, no próprio código do programa, de pontos ("gain points") onde o tempo ganho em execução possa ser detectado antes da conclusão da tarefa.

Vamos supor que a atividade que inclui as tarefas T_5 e T_7 é esporádica. Desta forma, no instante 5 temos:

$x_5(5) = 0+50-1 = 49$, $d_5(5) = 49+10 = 59$, $c_5(5) = 0$.

Observe que, como T_5 não possui sucessores remotos, o cálculo da sua folga usa o deadline e não o tempo máximo de resposta. A partir destes valores podemos calcular a folga da tarefa T_5 no instante 5. Observe que apenas a tarefa T_4 é capaz de interferir com a tarefa T_5.

$$S_5(5)=\left[d_5(5)-t-\sum_{\forall j\le 5} I_j\left(5,d_5(5)\right)\right]_0$$
$$S_5(5)=\left[59-5-I_4\left(5,d_5(5)\right)-I_5\left(5,d_5(5)\right)\right]_0$$

$$I_4(5,d_5(5)) = c_4(5) + f_4(5,d_5(5)) \times 1 + Min(1,(d_5(5) - x_4(5) - f_4(5,d_5(5)) \times P_4)_0)$$
$$I_4(5,59) = c_4(5) + f_4(5,59) \times 1 + Min(1,(59 - 67 - f_4(5,59) \times 67)_0)$$
$$f_4(5,59) = \lfloor (59 - 67) \div 67 \rfloor_0 = 0$$
$$I_4(5,59) = 0 + 0 \times 1 + Min(1,(59 - 67 - 0)_0) = 0 + 0 + 0 = 0$$
$$I_5(5,d_5(5)) = c_5(5) + f_5(5,d_5(5)) \times 5 + Min(5,(d_5(5) - x_5(5) - f_5(5,d_5(5)) \times P_5)_0)$$
$$I_5(5,59) = c_5(5) + f_5(5,59) \times 5 + Min(5,(59 - 49 - f_5(5,59) \times 50)_0)$$
$$f_5(5,59) = \lfloor (59 - 49) \div 50 \rfloor_0 = 0$$
$$I_5(5,59) = 0 + 0 \times 5 + Min(5,(59 - 49 - 0)_0) = 0 + 0 + 5 = 5$$
$$S_5(5) = [59 - 5 - 0 - 5]_0 = 49$$

Após as atualizações em função da conclusão da tarefa T_5, temos:

$S_4(5)=65$, $S_5(5)=49$, $S_6(5)=1$, $S_7(5)=1$, $S_{14}(5)=1$, $S_{17}(5)=1$.

Logo, neste momento é possível aceitar uma parte opcional no nível 6 com no máximo 1 unidade de tempo. Como foi suposto $O_6=1$, temos que é possível uma execução precisa de T_6, isto é, a parte opcional de T_6 pode ser aceita. Neste caso, a tabela de folgas é atualizada, ficando:

$S_4(5)=65$, $S_5(5)=49$, $S_6(5)=0$, $S_7(5)=0$, $S_{14}(5)=0$, $S_{17}(5)=0$.

5.6 Considerações Gerais Sobre o MDASS

É importante observar que várias fontes de capacidade ociosa presentes em um sistema são aproveitadas pelo MDASS. Por exemplo, sempre que uma tarefa com prioridade maior que $\rho(T_i)$ ocupa o processador por um tempo menor que o pior caso, ela gera sobre T_i uma interferência menor que aquela considerada para o cálculo do tempo máximo de resposta de T_i. Esta interferência já computada poderá ser então gerada por alguma parte opcional, sem alterar a escalonabilidade de T_i. O mesmo acontece quando uma tarefa é ativada com uma frequência menor que a máxima prevista.

O algoritmo MDASS foi implementado e diversos experimentos foram realizados. As estruturas de dados necessárias para implementar o MDASS são mínimas. Além das propriedades estáticas das tarefas (período, tempo máximo de execução, etc), é necessário apenas manter a quantidade de folga $S_i(t)$ de cada tarefa e o quanto cada tarefa já executou da presente liberação $c_i(t)$. No caso das tarefas esporádicas, é necessário também o instante da última liberação. Todos os demais valores podem ser calculados a partir destes.

As alterações realizadas no DASS para adapta-lo ao modelo de tarefas deste trabalho não aumentaram o seu custo computacional. As mudanças aconteceram nas equações e não no algoritmo de cálculo. Assim, as observações feitas em [DAV 93b] sobre o custo computacional do algoritmo DASS são válidas também para o MDASS.

A presença de relações de precedência entre tarefas alocadas a diferentes processadores tornaria possível, a princípio, a transferência de folga entre processadores. Considere duas tarefas T_i e T_j, onde $H(T_i) \neq H(T_j)$ e $dPred(T_j)=\{ T_i \}$. Nesta situação, é possível transferir folga entre T_i e T_j. Ao diminuir a folga de T_i estamos fazendo com que T_i receba menos interferência de partes opcionais durante sua execução em $H(T_i)$ e, portanto, termine antes. Como T_i termina antes, a execução de T_j pode iniciar antes e, portanto, T_j talvez possa receber mais interferência que o calculado inicialmente. Isto significa que a folga de T_j poderia aumentar e ser usada para a execução de partes opcionais em $H(T_j)$. Este raciocínio teria que ser elaborado para situações de múltiplas precedências. Também é possível realizar o caminho inverso e transferir folga de T_j para T_i. Neste caso, seria necessária a troca de mensagens entre as instâncias do teste de aceitação executando em $H(T_i)$ e $H(T_j)$. Na medida em que mensagens fossem utilizadas pelo próprio teste de aceitação, poderíamos ter transferência de folga entre processadores através da negociação do tempo máximo de resposta das tarefas com sucessores remotos. É uma questão em aberto o custo computacional da transferência de folga em ambiente distribuído.

Como descrito na seção 5.4, o MDASS preserva o tempo máximo de resposta das tarefas com sucessores remotos. A solução proposta neste trabalho também não emprega qualquer tipo de mensagem entre processadores com o propósito de aumentar a eficiência do método. Como definido, o MDASS não considera a possibilidade de transferência de folga entre processadores. Esta característica do MDASS foi uma consequência da opção por uma solução simples e com baixo custo computacional. Entretanto, existe amplo espaço para a investigação futura de novas soluções para a identificação de folgas em ambiente distribuído quando tarefas possuem relações de precedência.

5.7 Conclusões

Neste capítulo foi considerado o problema do teste de aceitação para partes opcionais quando a aplicação segue o modelo de tarefas descrito no capítulo 3. Inicialmente, foram descritas diversas abordagens presentes na literatura: baseadas em planejamento, baseadas em servidores, tomada estática de folga e tomada dinâmica de folga. Neste trabalho é adotada a abordagem da tomada dinâmica de folga aproximada (DASS).

O modelo de tarefas original do DASS difere daquele adotado por este trabalho. Logo, foi necessário descrever como o DASS pode ser utilizado em nosso contexto. Tarefas esporádicas com *jitter* de liberação recebem tratamento especial no momento do cálculo da folga. Tarefas com sucessores remotos precisam preservar o seu tempo máximo de resposta, para evitar que o atraso propagado através das relações de precedência faça alguma tarefa remota perder o seu deadline. A existência de dependência inter-tarefa pode ser aproveitada para aumentar a eficiência da detecção de folgas. O método resultante da modificação do DASS foi chamado de MDASS. As tabelas necessárias para a sua implementação em um dado processador são proporcionais ao número de tarefas alocadas ao processador em questão.

Como resultado do exposto neste capítulo, temos uma solução de escalonamento para a aceitação de partes opcionais em aplicações que seguem nosso modelo de tarefas. Em resumo, aplicações que incluem relações de precedência em ambiente distribuído.

6 Políticas de Admissão para Partes Opcionais

6.1 Introdução

Como descrito no capítulo 3, a abordagem adotada neste trabalho inclui a execução de partes opcionais quando o processador não estiver executando partes obrigatórias. É possível que, em determinados momentos, o conjunto de partes opcionais disponíveis para execução seja maior que aquilo que pode ser aceito. Então, uma política de admissão é usada para definir quais, entre as partes opcionais disponíveis, serão oferecidas ao teste de aceitação. As partes opcionais que passarem pela política de admissão e pelo teste de aceitação serão executadas. A política de admissão funciona como um filtro que, colocado antes do teste de aceitação, descarta as partes opcionais que acrescentam menos utilidade ao sistema.

Este trabalho apresenta duas novas heurísticas para serem usadas como política de admissão quando tarefas imprecisas são empregadas. Estas heurísticas deverão ser usadas em conjunto com testes "off-line" de escalonabilidade e testes "on-line" de aceitação. O teste de escalonabilidade, executado em tempo de projeto, garante que ao menos a parte obrigatória de cada tarefa será sempre completada antes do seu deadline. O teste de aceitação, empregado em tempo de execução, verifica se a execução de uma dada parte opcional vai ou não comprometer a execução das partes obrigatórias. O objetivo da política de admissão é maximizar a utilidade do sistema através da seleção de partes opcionais para a execução. Com respeito a este objetivo, as heurísticas propostas são analisadas através de simulação e comparadas com duas outras heurísticas presentes na literatura.

Neste trabalho é suposto que as tarefas imprecisas possuem dependências intra-tarefa e inter-tarefa. Este estudo se limita à técnica de múltiplas versões onde duas versões são usadas na programação de tarefas imprecisas. Quando múltiplas versões são usadas, é necessário decidir qual versão de uma tarefa executará antes da tarefa começar. Uma vez que a tarefa iniciou sua execução não é mais possível rever aquela decisão. Temos também que a parte opcional será completamente executada (quando a versão primária é escolhida) ou não será executada (quando a versão secundária é escolhida). É dito neste caso que a tarefa apresenta uma restrição 0/1.

O restante do capítulo está organizado da seguinte forma: a seção 6.2 descreve o problema de escalonamento a ser resolvido. A seção 6.3 descreve as heurísticas propostas neste trabalho como políticas de admissão. Descreve também duas heurísticas conhecidas que serão usadas para fins de comparação. A seção 6.4 descreve como foram feitas as simulações que avaliam a capacidade das diferentes políticas de admissão de aumentar o valor total do sistema onde são empregadas. A seção 6.5 mostra os resultados das simulações. Finalmente, na seção 6.6 são feitos os comentários finais sobre as políticas de admissão apresentadas.

6.2 Formulação do Problema

As heurísticas apresentadas neste capítulo deverão ser usadas como política de admissão em um sistema segundo o modelo de tarefas descrito no capítulo 3. É suposto que

cada tarefa T_i está associada com um valor nominal V_i. Este valor nominal não considera as dependências entre tarefas. A forma como valores nominais são atribuidos às tarefas constitui um problema de especificação e projeto de sistemas em software e foge do escopo deste trabalho.

Em tempo de execução, a cada liberação $T_{i,k}$ da tarefa T_i é associado um valor efetivo $V_{i,k}$. O valor efetivo é calculado a partir do valor nominal e das dependências entre tarefas existentes na aplicação. O valor adicionado por $T_{i,k}$ ao sistema será zero no caso de uma execução imprecisa ou $V_{i,k}$ quando a parte opcional for completamente executada. O objetivo geral da solução de escalonamento deve ser maximizar o valor total do sistema, dado pelo somatório dos valores adicionados por todas as liberações de todas as tarefas.

A dependência intra-tarefa entre as liberações $T_{i,k}$ e $T_{i,k+1}$, da tarefa T_i, é modelada assumindo-se que uma execução imprecisa de $T_{i,k}$ vai aumentar o valor efetivo de uma execução precisa de $T_{i,k+1}$. O valor efetivo da liberação $T_{i,k+1}$ será:

V_i	quando a execução de $T_{i,k}$ é precisa;
$V_i + (\alpha_i . V_{i,k})$	quando a execução de $T_{i,k}$ é imprecisa;

onde α_i é a taxa de recuperação da tarefa T_i.

A dependência inter-tarefa entre as liberações $T_{j,k}$ e $T_{i,k}$ é modelada assumindo-se que uma execução precisa de $T_{j,k}$ reduzirá o tempo de execução no pior caso das partes obrigatória e opcional de $T_{i,k}$. Em outras palavras, o tempo de execução de $T_{i,k}$ no pior caso será:

$M_i + O_i$	quando a execução de $T_{j,k}$ é imprecisa;
$\beta_{j,i} . M_i + \gamma_{j,i} . O_i$	quando a execução de $T_{j,k}$ é precisa;

onde $\beta_{j,i}$ e $\gamma_{j,i}$, $0<\beta_{j,i}\leq 1$, $0<\gamma_{j,i}\leq 1$, são os fatores de redução ligando T_j a T_i.

Como foi destacado no capítulo 3, nenhum dos dois tipos de dependência definidos é capaz de aumentar a carga obrigatória do sistema. A dependência intra-tarefa resulta em aumento no valor efetivo de algumas partes opcionais. A dependência inter-tarefas resulta na redução do tempo máximo de execução de algumas tarefas. De qualquer modo, a carga obrigatória do sistema não aumenta.

O valor total do sistema até o instante **t** será denotado por V(t). O valor $\Phi(t)$ representa o tempo total de processador, dentro do intervalo [0,t), que não foi usado para executar partes obrigatórias. A densidade média de valor obtida pelo sistema até o instante **t** será denotada por $\Lambda(t)$ e calculada da seguinte forma:

$$\Lambda(t) = V(t) / \Phi(t) .$$

Os valores V(t), $\Phi(t)$ e $\Lambda(t)$ haviam sido definidos antes em [DAV 95] para ambiente monoprocessado. Nesta tese esta definição será ampliada para o ambiente distribuído. No caso de um processador **p** específico, $V_p(t)$ denota o valor total das tarefas executadas no processador **p** até o instante **t**. De forma semelhante, será empregado $\Phi_p(t)$ para denotar o tempo disponível para partes opcionais neste processador dentro do intervalo [0,t). A densidade média de valor observada no processador **p**, $\Lambda_p(t)$, é dada por:

$\Lambda_p(t) = V_p(t) / \Phi_p(t)$.

Como definido no capítulo 3, a densidade de valor $\lambda_{i,k}$, associada com a parte opcional da liberação $T_{i,k}$ da tarefa T_i, é dada pela divisão de seu valor efetivo pelo seu tempo de execução no pior caso, após as correções em função das dependências inter-tarefa. Em outras palavras, temos:

$$\lambda_{i,k} = \frac{V_{i,k}}{O_i \times \prod_{\forall j.T_j \in [d\,Pred(T_i) \cap Opcio(T_{j,k})]} \gamma_{j,i}},$$

onde $Opcio(T_{j,k})$ representa o conjunto de tarefas cuja k-ésima liberação foi executada completamente. O cálculo de $\lambda_{i,k}$ considera o tempo máximo de execução da parte opcional de T_i após as correções devido às dependências inter-tarefa.

6.3 Descrição das Heurísticas

Nesta seção serão descritas diversas heurísticas que podem ser empregadas como política de admissão no modelo de tarefas considerado. As duas primeiras políticas (FCFS e AVDT) foram descritas em [DAV 95] e serão simuladas para fins de comparação. As outras duas políticas são originais e sua eficiência em aumentar o valor total do sistema será comparada com a eficiência das duas primeiras heurísticas em diferentes tipos de carga.

6.3.1 Ordem de Chegada (FCFS)

Nesta política todas as partes opcionais são sempre consideradas para execução. Quando uma tarefa vai iniciar sua execução, o teste de aceitação é sempre aplicado sobre sua parte opcional. Caso ela seja aceita, então a versão precisa da tarefa é executada. Note que esta política de admissão não considera o fato das tarefas possuirem diferentes valores, refletindo diferentes graus de utilidade para o sistema. Embora o nome "Admite Todos" fosse mais apropriado para esta política, o nome "Ordem de Chegada" (FCFS) será usado para manter compatibilidade com a literatura existente.

6.3.2 Adaptive Value Density Threshold (AVDT)

Nesta política o suporte de execução mantém atualizado o valor $\Lambda(t)$, ou seja, a densidade média de valor obtida pelo sistema até o momento. O valor $\Lambda(t)$ é usado como um limite mínimo para a densidade de valor das partes opcionais. Somente partes opcionais que possuem uma densidade de valor maior ou igual a $\Lambda(t)$ serão consideradas. Deste conjunto, aquelas que passarem no teste de aceitação serão executadas.

A heurística AVDT foi originalmente definida para um modelo de tarefas onde não existe dependência de qualquer tipo entre as tarefas. Desta forma, a densidade de valor associada com uma tarefa é simplesmente seu valor nominal dividido pelo tempo de execução no pior caso da parte opcional. Neste trabalho vamos adaptar AVDT ao nosso modelo de tarefas e usar a densidade de valor $\lambda_{i,k}$, como definida na seção anterior. Desta forma, a idéia básica do AVDT é mantida. Observe que as informações necessárias para

calcular $\lambda_{i,k}$ estão sempre disponíveis no momento que uma tarefa inicia sua execução, ou seja, no momento em que a heurística é empregada.

Como a política AVDT foi originalmente proposta para ambiente centralizado e monoprocessado, ela emprega o valor $\Lambda(t)$. Entretanto, em um sistema distribuído, o valor atual de $\Lambda(t)$ é uma informação global do sistema e depende de informações espalhadas pelos diversos processadores. A necessidade de usar esta informação global tornaria pouco atraente o uso desta política. Além disto, a decisão de admitir ou não uma dada parte opcional é uma decisão local. Ela deve considerar a densidade média de valor observada no processador em questão, e não a densidade média de valor do sistema como um todo. Desta forma, vamos adaptar a política AVDT para um contexto distribuído e aceitar as partes opcionais sempre que $\lambda_{i,k} \geq \Lambda_p(t)$, onde $p=H(T_i)$.

6.3.3 Compensated Value Density Threshold (CVDT)

Esta política de admissão, introduzida no presente texto, emprega o mesmo princípio do AVDT. Como antes, somente partes opcionais que possuem densidade de valor $\lambda_{i,k}$ superior a um determinado limite mínimo serão consideradas para execução. Deste conjunto, aquelas que passarem no teste de aceitação serão executadas. Entretanto, o limite mínimo utilizado é diferente daquele empregado no AVDT.

O limite mínimo utilizado no CVDT, denotado por ζ, é dado por:

$$\zeta = \Lambda_p(t) \times Min(5 \times \pi_p(t), 1.1), \qquad \text{[Equação 6.1]}$$

onde $p=H(T_i)$ e $\pi_p(t)$ representa a taxa de rejeição do teste de aceitação no processador **p**, ou seja, o número de partes opcionais rejeitadas dividido pelo número de partes opcionais submetidas ao teste de aceitação no processador **p**.

Como antes, a densidade média de valor $\Lambda_p(t)$, obtida pelo processador **p**, é usada. Mas agora ela é multiplicada por um fator de correção que leva em consideração a taxa de rejeição do teste de aceitação. Quando a taxa de rejeição for 20%, o próprio $\Lambda_p(t)$ é usado. Quando a taxa de rejeição for maior que 20%, indicando que a política de admissão está deixando passar muitos candidatos, o valor limite aumenta. Quando a taxa de rejeição é menor que 20%, indicando portanto que o teste de aceitação está recebendo poucos candidatos, o valor limite diminui. O valor 1.1 aparece na [Equação 6.1] apenas para limitar o efeito do multiplicador. Desta forma, o fator de correção poderá variar entre zero (quando $\pi_p(t)=0$) e 1.1 (quando $5 \times \pi_p(t) \geq 1.1$).

Os valores 1.1 como limitante e 20% como taxa de rejeição ideal foram escolhidos com base em diversas experiências. É possível que, para diferentes tipos de aplicações, um ajuste fino destes valores resultasse em um maior valor total do sistema.

6.3.4 Compensated Value Density Threshold for Inter-Task Dependencies (INTER)

Esta política, também introduzida no presente texto, é semelhante a anterior no sentido que compara uma densidade de valor com um limite mínimo. Como antes, somente partes opcionais que possuem uma densidade superior ao limite mínimo serão consideradas

para execução. A política INTER emprega o mesmo limite mínimo ζ para as densidades, dado pela [Equação 6.1].

No momento que a liberação $T_{i,k}$ vai iniciar sua execução, a política INTER compara o limite mínimo ζ com o resultado da soma da sua densidade de valor $\lambda_{i,k}$ com o fator de correção $\varepsilon_{i,k}$. A parte opcional será admitida quando $\lambda_{i,k} + \varepsilon_{i,k} \geq \zeta$.

Na definição do fator de correção $\varepsilon_{i,k}$ é considerado o fato de uma execução precisa de $T_{i,k}$ reduzir o tempo de execução obrigatória das suas tarefas sucessoras. O valor que uma execução precisa de $T_{i,k}$ contribui para o sistema é ampliado pela provável execução futura de outras partes opcionais. Estas futuras execuções aproveitariam a redução no tempo máximo de execução obrigatória das tarefas sucessoras de $T_{i,k}$. No cálculo de $\varepsilon_{i,k}$ é suposta uma densidade de valor $0.5\times\zeta$ para tais futuras execuções.

A definição de $\varepsilon_{i,k}$ leva também em consideração o fato de uma tarefa poder possuir mais de um predecessor. O tempo máximo de execução obrigatória $m_j(t)$, relativo a liberação $T_{j,k}$, inclui as correções devido aos predecessores de T_j que já executaram precisamente. Ele é calculado da seguinte forma:

$$m_j(t) = M_j \times \prod_{\forall h.T_h \in d\,Pred(T_j) \cap Opcio(T_{j,k})} \beta_{h,j}$$

O fator de correção $\varepsilon_{i,k}$ é dado por:

$$\varepsilon_{i,k} = \frac{0.5\times\zeta\times\sum_{\forall j.T_j \in dSuc(T_i)}[(1-\beta_{i,j})\times m_j(t)]}{O_i \times \prod_{\forall h.T_h \in d\,Pred(T_i) \cap Opcio(T_{i,k})} \gamma_{h,i}}$$

onde ζ é o limite mínimo definido antes e $Opcio(T_{j,k})$ e $Opcio(T_{i,k})$ denotam o conjunto de tarefas que tiveram sua k-ésima liberação executada precisamente e, desta forma, podem afetar o tempo máximo de execução das tarefas T_j e T_i, respectivamente.

É importante observar que, para efeito da política de admissão, o cálculo de $m_j(t)$ é aproximado. Considere a aplicação descrita pela figura 6.1. Suponha que as tarefas T_1, T_2 e T_0 são liberadas simultaneamente no instante zero. No processador 2 a tarefa T_0, mais prioritária, inicia a sua execução. No processador 1 a tarefa única T_1 também vai iniciar sua execução. É necessário determinar neste momento se a execução de T_1 será precisa ou não. Para tanto, é necessário termos $\lambda_{1,0} + \varepsilon_{1,0} \geq \zeta$.

Para determinar o valor de $\varepsilon_{1,0}$ é necessário determinar o valor de $m_3(0)$. Entretanto, no instante zero ainda não é possível afirmar se a tarefa T_2 será ou não executada precisamente. Uma execução precisa de T_2 reduziria o tempo máximo de execução obrigatória de T_3, reduzindo $m_3(t)$ e também a vantagem de executar precisamente T_1. Nestes casos, a política supõe que a execução de T_2 será imprecisa. Em outras palavras, toda redução de tempo de execução não conhecida é suposta não existente.

Efeito semelhante é causado pela distribuição. Suponha que, no instante em que T_1 inicia sua execução, T_2 também já iniciou. Neste caso, o fato de T_2 estar sendo executado de forma precisa ou não é conhecido no processador 2, mas não é conhecido no

processador 1. Neste caso, novamente a política supõe que as reduções desconhecidas são inexistentes.

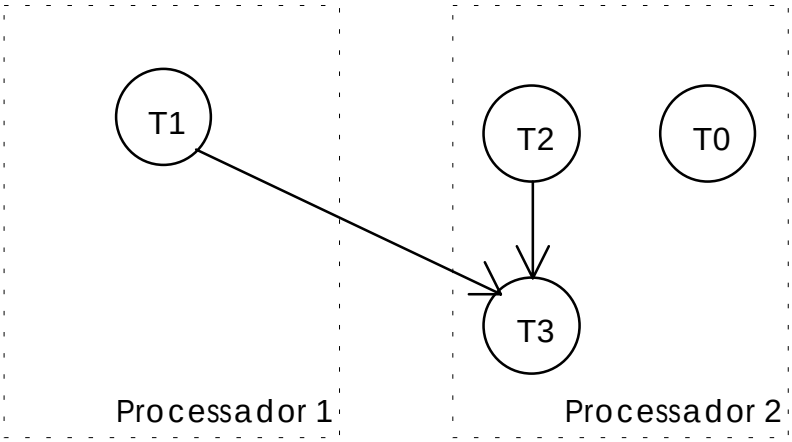

Figura 6.1 - Aplicação hipotética composta por 4 tarefas e 2 processadores.

6.4 Simulação

As heurísticas descritas na seção anterior são avaliadas através de simulação. As simulações realizadas procuram determinar a eficiência das heurísticas no sentido da sua capacidade de aumentar o valor total do sistema simulado. Desta forma, a eficiência das heurísticas propostas neste artigo será comparado com a eficiência de duas políticas existentes, isto é, FCFS e AVDT.

As simulações são realizadas a partir de aplicações compostas por conjuntos de tarefas gerados aleatoriamente, dentro do perfil descrito a seguir. Cada aplicação é alocada segundo o algoritmo que será apresentado no capítulo 7. A escalonabilidade de uma alocação em particular, no que diz respeito às partes obrigatórias, é testada segundo o algoritmo apresentado no capítulo 4. Uma vez definida a alocação e garantido que as partes obrigatórias podem ser executadas dentro do deadline, a execução da aplicação é simulada. Durante a simulação o algoritmo DASS modificado, apresentado no capítulo 5, é usado como teste de aceitação. Cada uma das políticas apresentadas na seção anterior é empregada, separadamente, como política de admissão. O valor total gerado pela aplicação é então medido.

A carga de tarefas foi definida de maneira semelhante a outros trabalhos presentes na literatura ([AUD 94b], [DAV 95]). Para efeito de simulação, foi desprezado o tempo de processador ("overhead") gasto devido aos testes de aceitação e a política de admissão. É importante notar que todas as políticas de admissão simuladas empregam o mesmo teste de aceitação. Logo, este custo ("overhead") é o mesmo para todas.

Todos os experimentos foram feitos a partir de aplicações compostas por conjuntos de 40 tarefas, executando em 4 processadores. A exata distribuição de tarefas por processadores é determinada pelo algoritmo de alocação, não sendo necessariamente sempre 10 tarefas por processador. Em cada conjunto de tarefas, 13 tarefas possuem período entre 30 e 300 ticks, outras 13 possuem período entre 300 e 3000 e as 14 tarefas restantes possuem período entre 3000 e 30000. Cada aplicação inclui ainda 15 relações de precedência sorteadas aleatoriamente. Os períodos das tarefas são ajustados de forma que

todas as tarefas pertencentes a uma mesma atividade possuam períodos iguais. Da mesma forma, todas as tarefas iniciais de uma mesma atividade possuem o mesmo *jitter* máximo de liberação, sorteado aleatoriamente entre 0 e 10 ticks.

Os deadlines das tarefas foram gerados de forma aleatória, a partir de uma distribuição uniforme entre 30 e o período da tarefa em questão. Os deadlines são ainda acertados para que eles sejam sempre crescentes ao longo das relações de precedência. Assim, deadline monotônico pode ser utilizado para definir as prioridades das tarefas.

Os tempos máximos de computação obrigatória foram gerados também aleatoriamente, mas de maneira que a carga obrigatória total seja igual a 20%, 50% ou 80% da capacidade total do sistema. Por exemplo, considerando que o sistema dispõe de 4 processadores, uma carga correspondente a 50% da capacidade total do sistema é equivalente a uma carga de 200% aplicada sobre um único processador. A carga obrigatória exata alocada a cada processador é determinada pelo algoritmo de alocação.

Os tempos máximos de computação opcional foram igualmente gerados de forma aleatória, mas de tal sorte que a carga opcional total no sistema variasse de 30% a 300%. Não existe uma relação previamente definida entre o tempo máximo de execução da parte opcional O_i da tarefa T_i e o seu tempo máximo de execução obrigatória M_i. Como será visto no capítulo 7, o algoritmo de alocação empregado procura fazer o balanceamento da carga entre os processadores, obtendo bons resultados.

Para efeito de simulação, somente foram considerados conjuntos de tarefas cujas partes obrigatórias puderam ser garantidas antes da execução. Para cada combinação de carga obrigatória e opcional foram simulados 15 conjuntos de tarefas diferentes. Cada uma das políticas de admissão citadas na seção anterior foi aplicada aos mesmos 15 conjuntos de tarefas. Os resultados apresentados correspondem à média destas 15 execuções.

Os valores V_i das partes opcionais foram estabelecidos também aleatoriamente seguindo uma distribuição uniforme entre 0 e 10. Tanto as taxas de recuperação α_i quanto os fatores de redução $\beta_{i,j}$ e os fatores de redução $\gamma_{i,j}$ das tarefas foram escolhidos segundo uma distribuição uniforme entre 0 e 1.

Devido a forma como os parâmetros das tarefas são gerados, é possível a ocorrência de partes opcionais impossíveis de serem aceitas. São partes opcionais que, mesmo com a máxima redução possível devido à dependência inter-tarefa, são maiores que o próprio deadline da tarefa. É possível determinar, ainda em tempo de projeto, que tais partes opcionais jamais serão executadas. Este fenômeno acontece principalmente quando a carga total de opcionais do sistema chega perto de 300% da sua capacidade. Neste caso, o simulador automaticamente muda para zero o valor base destas tarefas. Isto equivale a retirar do sistema as partes opcionais que, ainda em projeto, são identificadas como impossíveis de serem executadas.

6.5 Resultados da Simulação

Nesta seção serão apresentados os resultados das simulações realizadas. As tabelas de 6.1 a 6.9 resumem estes resultados. A figura de mérito observada é o valor total obtido pela heurística em questão em relação ao valor total obtido por FCFS, para a mesma carga

de tarefas. Por exemplo, um valor 1.5 significa que a heurística em questão obteve 50% a mais de valor do que quando FCFS for usado para a mesma carga de tarefas.

O custo dos algoritmos não foi incluido nas simulações realizadas. Como dito antes, todos utilizam o mesmo teste de aceitação. Com respeito ao custo ("overhead") das políticas de admissão, é importante notar que os valores efetivos e os tempos máximos de execução podem ser mantidos pelo suporte de execução. Eles seriam atualizados sempre que uma tarefa é executada precisamente. Desta forma, todos os produtórios que aparecem nas equações já estariam calculados no momento de executar a política de admissão. Neste caso, FCFS, AVDT e CVDT teriam uma complexidade O(1). Já INTER possuiria uma complexidade O(N) em função do somatório que aparece no numerador. No caso extremo, todas as tarefas são sucessoras de T_i e podem ser afetadas por sua execução precisa. Em termos práticos, cada tarefa possui alguns poucos sucessores diretos e, portanto, o custo computacional da política INTER é similar ao das outras políticas.

As tabelas 6.1, 6.2 e 6.3 descrevem os resultados obtidos quando existe apenas dependência intra-tarefa. Nesta situação, CVDT e INTER apresentam o mesmo comportamento. Isto é esperado pois na falta de dependência inter-tarefa temos sempre que $\varepsilon_{i,k}=0$ e as duas heurísticas ficam iguais.

Em todas as combinações de carga apresentadas nas tabela 6.1 e 6.2 (carga obrigatória de 20% e 50%, respectivamente), CVDT e INTER foram superiores a FCFS e AVDT. O maior valor total associado com CVDT e INTER com relação a AVDT está na capacidade daqueles em detectar quando o processador está subutilizado e então baixar o limite mínimo de forma adequada. Esta superioridade torna-se menor a medida que a carga obrigatória aumenta. Este fato fica bem evidenciado pela tabela 6.3 (carga obrigatória de 80%). Os resultados apresentados por esta tabela mostram que CVDT e INTER são ligeiramente melhores que AVDT em cargas opcionais baixas, até 90%. Com carga obrigatória de 80% e carga opcional igual ou maior que 120%, a política AVDT mostrou maior eficiência com respeito ao valor total do sistema.

	Carga obrigatória de 20%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	0.980	0.971	0.966	1.010	1.072	1.145	1.229	1.267	1.289	1.336
CVDT	1.000	1.012	1.019	1.070	1.130	1.201	1.289	1.316	1.349	1.376
INTER	1.000	1.012	1.019	1.070	1.130	1.200	1.289	1.316	1.349	1.376

Tabela 6.1 - Carga obrigatória de 20% com interferência intra-tarefa.

	Carga obrigatória de 50%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	1.005	1.048	1.156	1.251	1.325	1.353	1.407	1.399	1.429	1.463
CVDT	1.026	1.073	1.182	1.276	1.366	1.377	1.445	1.437	1.475	1.500
INTER	1.026	1.073	1.182	1.276	1.367	1.378	1.446	1.437	1.475	1.500

Tabela 6.2 - Carga obrigatória de 50% com interferência intra-tarefa.

	Carga obrigatória de 80%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	1.158	1.479	1.657	1.764	1.827	1.780	1.656	1.593	1.543	1.541
CVDT	1.168	1.487	1.679	1.752	1.798	1.748	1.614	1.572	1.530	1.537
INTER	1.168	1.486	1.677	1.753	1.797	1.748	1.614	1.573	1.530	1.536

Tabela 6.3 - Carga obrigatória de 80% com interferência intra-tarefa.

As tabelas 6.4, 6.5 e 6.6 mostram os resultados da simulação quando apenas dependência inter-tarefa está presente. Novamente CVDT e INTER são mais eficientes do que o AVDT em detectar uma baixa utilização obrigatória. Isto resulta em CVDT e INTER gerarem mais valor que AVDT quando a carga obrigatória é de 20% ou 50% (tabelas 6.4 e 6.5, respectivamente). Quando a carga obrigatória é de 80% (tabela 6.6), as três políticas resultam em um valor total semelhante para os sistemas analisados, embora este valor seja bem superior àquele obtido por FCFS.

Embora as equações associadas com INTER considerem explicitamente a dependência inter-tarefa, esta heurística resulta em valores similares ao CVDT. Uma pequena superioridade de INTER sobre CVDT pode ser notada com carga obrigatória de 80% (tabela 6.6) e carga opcional entre 30% e 150%.

As simulações descritas em [OLI 96b] mostraram que a política INTER é superior para ambiente monoprocessado, na presença de dependência inter-tarefa. Entretanto, esta superioridade não aparece nas simulações de ambiente distribuído. A vantagem da política INTER sobre CVDT estaria em gerar "tempo extra de processador" aproveitando a economia gerada pelo fator de redução β. Esta economia surge quando uma tarefa com sucessores diretos é executada precisamente, reduzindo o tempo máximo de execução obrigatória destes sucessores. Entretanto, em ambiente distribuído, muitos destes sucessores diretos são remotos. Neste caso, o algoritmo de tomada de folga modificado não pode alterar o tempo de resposta da tarefa em questão, sob pena de violar o deadline das tarefas sucessoras. Esta característica da tomada de folga modificada reduz consideravelmente a folga das tarefas com sucessores, exatamente aquelas que a política INTER procura executar precisamente para gerar tempo de processador no futuro. O resultado é que a política INTER não consegue oferecer uma eficiência superior ao CVDT.

	Carga obrigatória de 20%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	0.972	0.953	0.973	1.031	1.085	1.152	1.267	1.354	1.435	1.510
CVDT	1.002	1.005	1.033	1.118	1.202	1.289	1.401	1.484	1.564	1.615
INTER	1.002	1.005	1.033	1.118	1.202	1.288	1.401	1.484	1.564	1.616

Tabela 6.4 - Carga obrigatória de 20% com interferência inter-tarefa.

	Carga obrigatória de 50%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	0.976	0.958	1.072	1.171	1.299	1.468	1.621	1.639	1.625	1.614
CVDT	1.003	1.026	1.151	1.280	1.375	1.552	1.678	1.685	1.661	1.622
INTER	1.003	1.025	1.153	1.281	1.374	1.550	1.677	1.685	1.662	1.622

Tabela 6.5 - Carga obrigatória de 50% com interferência inter-tarefa.

	Carga obrigatória de 80%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	1.072	1.363	1.637	1.758	1.782	1.941	1.847	1.762	1.723	1.702
CVDT	1.080	1.380	1.689	1.782	1.775	1.941	1.838	1.759	1.713	1.688
INTER	1.131	1.409	1.712	1.809	1.807	1.958	1.856	1.761	1.717	1.688

Tabela 6.6 - Carga obrigatória de 80% com interferência inter-tarefa.

As tabelas 6.7, 6.8 e 6.9 mostram os valores obtidos nas simulações quando existe simultaneamente dependência intra-tarefa e inter-tarefa. Os resultados são semelhantes aos obtidos com dependência inter-tarefa. Temos CVDT e INTER sempre com valor total gerado praticamente igual. Também, CVDT e INTER são sempre melhores que AVDT quando a carga obrigatória é de 20% ou 50%. As três heurísticas apresentam um valor total gerado semelhante com carga obrigatória de 80%.

	Carga obrigatória de 20%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	0.983	0.983	0.982	1.008	1.058	1.127	1.175	1.233	1.287	1.367
CVDT	1.000	1.011	1.022	1.063	1.121	1.188	1.229	1.280	1.332	1.400
INTER	1.000	1.011	1.022	1.063	1.121	1.188	1.229	1.279	1.332	1.400

Tabela 6.7 - Carga obrigatória de 20% com interferência intra-tarefa e inter-tarefa.

	Carga obrigatória de 50%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	0.985	1.015	1.111	1.131	1.236	1.291	1.359	1.415	1.421	1.466
CVDT	1.014	1.058	1.170	1.194	1.286	1.323	1.386	1.434	1.435	1.476
INTER	1.014	1.058	1.171	1.195	1.288	1.324	1.386	1.434	1.433	1.475

Tabela 6.8 - Carga obrigatória de 50% com interferência intra-tarefa e inter-tarefa.

	Carga obrigatória de 80%									
	Carga opcional:									
	30%	60%	90%	120%	150%	180%	210%	240%	270%	300%
AVDT	1.059	1.390	1.450	1.519	1.571	1.654	1.685	1.655	1.685	1.591
CVDT	1.139	1.397	1.439	1.527	1.572	1.596	1.626	1.636	1.670	1.586
INTER	1.144	1.413	1.447	1.527	1.576	1.606	1.631	1.640	1.676	1.593

Tabela 6.9 - Carga obrigatória de 80% com interferência intra-tarefa e inter-tarefa.

Não é possível uma comparação direta dos resultados apresentados acima com os resultados apresentados em [CHU 90] e [FEN 94]. Em [CHU 90] existia a necessidade de garantir que as partes opcionais de todas as tarefas seriam eventualmente executadas. Em [FEN 94], embora uma parte opcional pudesse não ser jamais executada, haveria então um aumento no tempo de execução de partes obrigatórias, as quais então teriam que ser necessariamente executadas. Em função das soluções de escalonamento apresentadas nesta tese, uma parte opcional com valor nominal baixo provavelmente jamais será executada em um sistema com sobrecarga. Entretanto, este comportamento é coerente com o objetivo de reservar as sobras de processador para as partes opcionais que mais utilidade adicionem ao sistema como um todo.

É possível fazer uma comparação direta entre as políticas propostas neste artigo e o AVDT. Ambas as políticas CVDT e INTER apresentam resultados melhores que AVDT quando a carga obrigatória é de 20% ou 50%. Quando a carga obrigatória é de 80%, CVDT e INTER são melhores em carga opcional baixa (aproximadamente até 100%), enquanto AVDT é melhor com carga opcional alta (mais que 100%). Entretanto, a diferença entre as políticas é pequena e até inexistente em muitos casos.

Os melhores resultados do CVDT em relação ao AVDT são devidos ao novo limite mínimo de densidade usado. A política AVDT emprega $\Lambda_p(t)$ como limite mínimo. Em cargas baixas, o AVDT rejeita partes opcionais com baixa densidade de valor mesmo que o processador esteja livre. O limite mínimo ζ, usado por CVDT e INTER, considera o grau de ocupação do processador.

A política INTER considera explicitamente as dependências inter-tarefa. Desta forma, ela tenderia a apresentar resultados melhores que AVDT e CVDT na presença deste tipo de dependência. Entretanto, em função do comportamento da tomada de folga modificada, esta vantagem não apareceu nas experiências realizadas.

6.6 Conclusões

Neste capítulo foram apresentadas duas novas heurísticas, próprias para serem empregadas como política de admissão em sistemas de tarefas imprecisas. Seu comportamento foi analisado através de simulação e comparado com outros dois algoritmos presentes na literatura. Foram considerados casos onde existem relações de dependência entre diferentes liberações de uma mesma tarefa e/ou dependência entre tarefas diferentes. Foram também consideradas cargas obrigatórias de 20%, 50% e 80%, juntamente com cargas opcionais que variaram de 30% a 300%. As heurísticas propostas apresentaram, em

muitas situações, uma maior eficiência que as existentes na literatura, no sentido que foram capazes de gerar um maior valor total para os sistemas simulados.

É importante notar a natureza dinâmica de diversos valores considerados pelas heurísticas, tais como, $\Lambda_p(t)$, $\pi_p(t)$, etc. Estes valores devem ser considerados apenas em um intervalo de tempo recente e não pela vida total do sistema. Desta forma seria melhor tratada uma situação de mudança no modo de operação da aplicação ("mode change"), em termos dos valores de suas tarefas.

As soluções empregadas neste capítulo para a questão da política de admissão consideram que o comportamento opcional do sistema não é conhecido de forma determinista mas sim probabilista. Neste sentido, elas são similares aos algoritmos de substituição de página empregados na implementação de memória virtual em sistemas operacionais. Na implementação de memória virtual, as decisões são tomadas segundo a suposição de que o futuro próximo será semelhante ao passado recente. A mesma suposição é adotada aqui com respeito aos valores que descrevem a história da aplicação e são, por sua vez, usados para decidir sobre a admissão ou não de uma dada parte opcional.

7 Alocação de Tarefas Imprecisas em Ambiente Distribuído

7.1 Introdução

Este capítulo trata do problema da alocação de tarefas imprecisas em um ambiente distribuído. Este problema pode ser descrito como composto por dois objetivos. O objetivo primário é alocar as tarefas imprecisas de tal forma que as partes obrigatórias possam ser garantidas em tempo de projeto. Para tanto, será empregado o teste de escalonabilidade descrito no capítulo 4. O objetivo secundário é alocar as tarefas imprecisas de tal forma que a chance das partes opcionais executarem seja maximizada. Em outras palavras, efetuar um balanceamento de carga que evite sobrecargas localizadas quando a carga completa (obrigatória + opcional) é considerada. Obviamente, o objetivo secundário não pode ser capaz de compensar uma falha em atender ao objetivo primário. Em outras palavras, um excelente balanceamento de carga jamais poderá compensar o fato de uma única tarefa perder ser deadline, no pior caso, por uma única unidade de tempo.

Em aplicações de tempo real geralmente não existe migração de tarefas, pois esta técnica demanda recursos tanto de processamento como de comunicação e pode tornar a tarefa sendo migrada temporariamente indisponível. Como cada tarefa será sempre executada no mesmo processador, aumenta a importância de uma alocação adequada em tempo de projeto.

Como descrito no capítulo 3, além dos objetivos primário e secundário existem outras restrições quanto a alocação. É possível destacar os seguintes tipos de restrições adicionais:

- Definido no projeto que determinada tarefa já está associada com um processador em particular.

- Definido no projeto que determinado subconjunto de tarefas deve ficar necessariamente no mesmo processador.

- Definido no projeto que determinado subconjunto de tarefas deve ficar necessariamente em processadores diferentes.

O algoritmo de alocação procura aumentar as chances de uma parte opcional ser escalonada em tempo de execução. Uma primeira forma de fazer isto é balancear a carga no sistema, considerando a duração total das tarefas. A partir desta abordagem simples, três melhorias imediatas podem ser feitas.

Para efeito de balanceamento de carga, é mais realista considerar o tempo médio de execução das tarefas e não o tempo máximo de execução. O tempo máximo de execução é necessário para prover garantias em cenários de pior caso. Entretanto, o balanceamento é uma abordagem aproximada baseada no comportamento médio das tarefas e não no seu comportamento de pior caso. O tempo médio de execução neste caso é mais realista.

Esporádicas são tratadas, no pior caso, como periódicas com período igual ao intervalo mínimo entre ativações. No caso do balanceamento da carga, este tratamento é

pessimista. Novamente, para efeitos de balanceamento de carga, as esporádicas devem ser tratadas como periódicas com período igual ao intervalo médio entre ativações. Este tratamento fornece um cenário mais próximo do que acontece em tempo de execução.

Quando as tarefas possuem valores nominais fixos, é possível utilizar estes valores nominais para julgar o balanceamento de carga obtido. O objetivo de balancear carga nos processadores deveria ser acompanhado do objetivo de balancear o valor nominal da carga nos processadores. Duas situações são indesejáveis: termos opcionais demais em um mesmo processador e termos tarefas de muito valor concentradas em um mesmo processador. Desta forma, o objetivo de balancear a carga medida em termos de ocupação de processador deve ser acompanhado do objetivo de balancear a ocupação de processador ponderada pelo valor nominal das tarefas. Em aplicações onde o valor nominal de uma tarefa varia ao longo do tempo não é possível este tipo de análise. Neste caso, um bom balanceamento ponderado de carga em um dado instante pode se transformar em um péssimo balanceamento no instante seguinte, quando o valor nominal das tarefas muda. A forma como valores nominais são atribuidos às tarefas constitui um problema de especificação e projeto de sistemas em software e foge do escopo deste trabalho.

7.2 Revisão da Literatura

A maioria dos trabalhos publicados sobre Computação Imprecisa tratam apenas do escalonamento local. A influência da alocação das tarefas sobre a escalonabilidade das partes opcionais não é normalmente considerada. Uma exceção é [YU 92], que apresenta uma solução para o problema de alocação e escalonamento de tarefas imprecisas replicadas em um multiprocessador. Em tempo de projeto, heurísticas são empregadas para alocar as réplicas aos processadores e fixar qual será o tempo de processador total para cada tarefa. As tarefas são periódicas, independentes e possuem uma parte opcional do tipo "função monotônica". Localmente é empregado EDF ("Earliest Deadline First", [LIU 73]) ou RM ("Rate-Monotonic", [LIU 73]) para escalonar cada processador. É importante notar que, nesta proposta, o tempo de processador que cada parte opcional efetivamente recebe é definido ainda em projeto. Computação Imprecisa não é empregada para aumentar a adaptabilidade do sistema em tempo de execução. Ela é empregada para conseguir uma solução de projeto que garanta deadlines em um contexto de recursos limitados. Por exemplo, as folgas resultantes de um tempo de execução médio menor do que o tempo de execução no pior caso não são aproveitadas para executar partes opcionais.

A literatura sobre alocação de tarefas não imprecisas com restrições de tempo real em ambientes distribuídos é bastante extensa. Em função da complexidade exponencial do problema de alocação ([PAR 82a], [PAR 82b]), as soluções propostas tendem a ser subótimas. Em linhas gerais, podemos dividir as propostas em duas: pesquisa heurística e pesquisa aleatória. Na pesquisa heurística, o espaço de soluções é organizado na forma de uma árvore. O problema da alocação é resolvido através da identificação de um caminho nesta árvore. A identificação deste caminho é dirigida por uma heurística que procura sempre optar por alocações com melhores resultados. Quando fica claro que o caminho escolhido não levará a uma solução, é feito um retorno ("backtracking") e outro caminho é testado. Soluções baseadas em heurísticas podem ser encontradas, por exemplo, em [VER 91], [HOU 92a] e [BUR 95].

Determinados problemas permitem a construção de boas heurísticas, que conduzem rapidamente a uma solução. Mas em situações nas quais o problema de alocação se torna mais complexo, fica muito difícil criar heurísticas que capturem todas as sutilezas e implicações ("tradeoffs") das decisões tomadas. Neste momento, a pesquisa aleatória passa a ser atrativa. Na pesquisa aleatória não são empregadas heurísticas, mas sim uma função que avalia a qualidade de uma alocação. A partir de uma alocação insatisfatória, são realizadas mudanças aleatórias e a nova alocação é testada.

Enquanto algoritmos baseados em heurísticas exigem uma descrição implícita de "como construir uma solução", o recozimento simulado e os algoritmos genéticos exigem basicamente uma função para "avaliar a qualidade de uma solução". Isto é uma enorme vantagem quando o problema em questão possui diversos objetivos simultaneos, muitas vezes conflitantes entre si. Por exemplo, a melhor alocação para o sistema pode ser colocar no mesmo processador duas tarefas que, por definição no projeto, não podem executar no mesmo processador.

Um método interessante é o recozimento simulado ("simulated annealing", [KIR 83]). Propostas que empregam recozimento simulado para resolver o problema de alocação de tarefas em sistemas de tempo real podem ser encontradas em [TIN 92a] e [BUR 93]. Na mesma linha de pesquisa aleatória estão os algoritmos genéticos ([HOU 92b]).

7.3 Abordagem Adotada

O problema de alocação de tarefas em ambiente distribuído é bastante complexo. Na maioria dos casos, obter uma solução ótima implica em um custo de processamento proibitivo. Desta forma, são utilizadas abordagens subótimas. Uma forma de resolver o problema é através de pesquisa heurística sobre o espaço de soluções. Entretanto, na medida em que o problema se torna complexo, fica difícil sintetizar todos os seus aspectos em uma heurística simples. Por isto, será empregado neste trabalho um método de pesquisa aleatória chamado recozimento simulado ("simulated annealing", [KIR 83], [TIN 92a]).

Neste trabalho optamos pelo uso da técnica do recozimento simulado no problema de alocação de tarefas com restrições de tempo real em ambiente distribuído. O objetivo primário da alocação é garantir os deadlines das tarefas, considerando só as partes obrigatórias. O objetivo secundário será prover um razoável balançeamento de carga, de modo a evitar sobrecargas localizadas e aumentar as chances das partes opcionais executarem. Para isto, é necessário adaptar a função que avalia a qualidade de uma solução. As restrições de alocação definidas em projeto podem ser tratadas da mesma forma como são tratadas em [TIN 92a], e não serão mais consideradas neste trabalho.

7.3.1 Descrição Geral do Recozimento Simulado

O recozimento simulado é uma técnica de otimização que busca o ponto de menor energia em um espaço. A partir de uma solução inicial, é feita uma caminhada na vizinhança, em direção a pontos de energia menor. Esta caminhada leva naturamente a mínimos locais. Também são feitos saltos aleatórios, buscando regiões do espaço que não seriam atingidas através de uma caminhada buscando pontos vizinhos com energias menores. O salto aleatório permite que a pesquisa fuja do mínimo local, procurando

encontrar outros mínimos locais ainda menores ou até mesmo o mínimo global. Uma descrição detalhada da técnica em si pode ser encontrada em [KIR 83]. Um exemplo de como esta técnica pode ser empregada para solucionar o problema de alocação de tarefas em ambiente distribuído pode ser encontrado em [TIN 92a].

O algoritmo mostrado na tabela 7.1 resume a operação do recozimento simulado. O algoritmo apresentado é genérico. Ele pode resolver qualquer problema de otimização. Para uma aplicação específica é necessário definir o significado exato das várias funções empregadas.

Cada ponto do espaço pesquisado corresponde a uma solução para o problema de otimização em questão. No nosso caso, uma solução em particular inclui a associação de cada tarefa com um processador. O ponto aleatório inicial, a partir do qual a caminhada inicia, corresponde a uma solução onde as alocações são determinadas aleatoriamente.

```
i:= 0
j:= 0
Escolhe um ponto aleatório inicial Pi
Escolhe uma temperatura inicial Tj
Repete
        Repete
                Escolhe um ponto aleatório Q = Vizinho( Pi )
                Se Energia(Q) < Energia(Pi)
                        então  Pi+1 = Q

                        senão x = (Energia(Pi) − Energia(Q)) / Tj

                                Se e^x > Aleatório(0,1)
                                        então  Pi+1 = Q
                                        senão Pi+1 = Pi
                        i = i + 1
        Até Equilibrio( )
        Tj+1 = ReduzTemp( Tj )
        j = j + 1
Até Termina( )
```

Tabela 7.1 - Algoritmo básico do recozimento simulado.

No problema em questão, uma solução Q vizinha da solução P será obtida através de uma entre duas formas:

- Uma tarefa é escolhida aleatoriamente e movida para um processador também aleatório;
- Duas tarefas são escolhidas aleatoriamente e trocam de processador.

A função Energia é o aspecto mais importante do algoritmo. Como o recozimento simulado busca pontos de menor energia, a função Energia deve ser tal que quanto melhor uma solução, menor a energia do ponto correspondente. Esta função resume o problema

tratado. A construção de uma função Energia apropriada para o nosso problema será mostrada na próxima seção.

A função Aleatório(0,1) retorna uma valor aleatório, entre 0 e 1, seguindo uma distribuição de probabilidades uniforme. A comparação do valor gerado por esta função com e^x vai decidir se é realizado ou não o salto para um ponto de energia mais alta. Estes saltos permitem fugir de um mínimo local e buscar soluções melhores. Eles são controlados pela temperatura do sistema Tj. No início a temperatura é alta e qualquer salto é permitido. A medida que o tempo passa, a temperatura diminui e fica mais improvável um salto para pontos de maior energia.

A temperatura inicial deve ser tal que qualquer salto proposto seja aceito. Uma forma simples de chegar a este valor é partir de um valor pequeno para a temperatura e dobra-lo até que a taxa de saltos propostos aceitos seja próxima de 100%. Sua ordem de grandeza vai depender da ordem de grandeza dos valores retornados pela função Energia. É importante observar que uma temperatura inicial excessivamente alta torna o algoritmo mais lento, porém não altera a qualidade do resultado obtido.

A função Equilíbrio indica o momento de reduzir a temperatura, reduzindo a chance de um salto para pontos de energia maior. A forma usual para determinar este momento é estabelecer um limite para o número de saltos para pontos de menor energia e também para o número de tentativas de saltos. Quando um destes limites é atingido, o sistema é considerado em equilíbrio térmico. A função Equilíbrio afeta o tempo de execução do algoritmo na medida que é ela quem determina o momento de reduzir a temperatura do sistema. Entretanto, uma função equilíbrio muito "rápida" poderá reduzir a temperatura prematuramente e impedir que o sistema encontre o mínimo global.

A função ReduzTemp, responsável pela redução da temperatura., pode ser simplesmente ReduzTemp(T) = $\alpha \times$ T, onde $0 < \alpha < 1$. É importante notar que existem duas forças controlando a definição de α. Por um lado, valores menores para α fazem com que a temperatura caia rapidamente e a execução do algoritmo seja mais rápida. Por outro lado, para que o recozimento simulado encontre os pontos de mínimo global é necessário que o "resfriamento do sistema" seja lento. Isto é, que o algoritmo tenha tempo de buscar os mínimos através de seu caminhamento aleatório. Desta forma, um valor muito pequeno para α resultará em uma solução final de menor qualidade. A escolha de α, da mesma forma que a escolha da temperatura inicial, é através de experimentação. Uma forma simples é inicialmente executar o algoritmo com $\alpha = 0.99$ para um dado sistema. Em seguida o valor de α é reduzido e o algoritmo novamente executado para o mesmo sistema. O valor de α poderá ser reduzido enquanto a solução final não for afetada. Este valor final de α poderá ser utilizado nas futuras aplicações do algoritmo para problemas semelhantes ao problema de teste.

A função Termina decide quando a busca deve terminar. Uma forma simples é terminar a busca quando a solução se mantém a mesma (nenhum salto ocorre) após um determinado número de iterações. Por exemplo, após várias reduções de temperatura sem nenhuma redução na energia do sistema. A função Termina obviamente afeta o tempo de execução do algoritmo. Da mesma forma que a função Equilíbrio, uma função Termina muito "rápida" poderá terminar a execução do algoritmo prematuramente e impedir que o sistema encontre o mínimo global.

Como pode ser observado pela descrição do algoritmo, a aplicação do recozimento simulado exige uma etapa de calibração das constantes envolvidas. Os valores "temperatura inicial" e "fator α de redução da temperatura", assim como os critétios para "equilíbrio térmico" e para "terminação" estão relacionados entre si. Todos os quatro estão sujeitos a dois tipos de consideração. Por um lado, devem permitir que o recozimento simulado evolua de forma lenta o bastante para que um mínimo quase global seja encontrado. Por outro lado, não devem tornar a execução do algoritmo mais lenta do que o necessário para a localização do mínimo global. Em geral a calibração é feita a partir de um problema exemplo e depois os valores são usados novamente em problemas semelhantes.

7.3.2 A Função Energia

A função Energia empregada deve contemplar os objetivos primário e secundário da alocação. Como dito antes, restrições de alocação definidas em projeto podem ser tratadas da mesma forma que em [TIN 92a] e não serão consideradas neste trabalho. Da mesma forma, não serão considerados neste trabalho aspectos ligados à comunicação entre os processadores. O modelo de tarefas descrito no capítulo 3 supõe que, para qualquer solução de alocação empregada, o protocolo de comunicação utilizado garante um tempo máximo Δ para o envio de uma mensagem entre dois processadores distintos. Protocolos de comunicação em tempo real fogem do escopo deste trabalho.

A função Energia deverá ser uma composição dos seguintes termos:

- Energia associada com a escalonabilidade das tarefas, E_e;
- Energia associada com o balanceamento da carga no sistema, E_b.

Assim, a energia **E** associada com uma solução pode ser calculada por:

$$E = K_e \cdot E_e + K_b \cdot E_b$$

Os valores K_e e K_b definem os pesos das energias E_e e E_b na composição da energia total de uma dada solução de alocação. Além disto, estas duas constantes permitem que a medida de energia seja colocada no intervalo de valores desejado.

O emprego dos valores K_e e K_b permite que o aspecto escalonabilidade receba uma importância maior do que o aspecto balanceamento da carga. Jamais uma solução bem balanceada porém não escalonável terá uma energia menor do que uma solução com péssimo balanceamento porém com todos os deadlines garantidos. Ao escolhermos o valor de K_e suficientemente maior do que o valor de K_b estaremos garantindo que a escalonabilidade será o objetivo primário da alocação, enquanto o balanceamento é apenas um objetivo secundário. É possível afirmar que o algoritmo de alocação busca, entre todas as soluções que garantem a escalonabilidade do sistema, aquela solução com o melhor balanceamento de carga.

O valor E_e é uma medida do quanto deadlines estão sendo perdidos, ou seja, do quanto o sistema está longe de ser garantido. O fato da análise de escalonabilidade apresentada no capítulo 4 retornar o tempo de respostas das tarefas é aproveitado. Supondo que a análise de escalonabilidade determina o tempo de resposta máximo R_i para cada tarefa T_i com deadline D_i, a energia é calculada por:

$$E_e = \sum_{i=1}^{n} \text{Max}(0, R_i - D_i)$$

Observe que, na equação, o fato do tempo de resposta de uma tarefa ser inferior ao seu deadline não diminui a energia da solução. Isto reflete o fato de não ser necessário minimizar os tempos de resposta, mas apenas garantir os deadlines.

Serão feitas duas adições ao modelo de tarefas proposto, ampliando a descrição de cada tarefa da aplicação. Estas duas adições visam melhorar a alocação no que diz respeito ao aspecto balanceamento de carga. É importante notar que elas serão utilizadas apenas no cálculo da energia E_b. O aspecto escalonabilidade, representado pela energia E_e, deve necessariamente considerar os cenários de pior caso. As duas adicões são:

- Cada tarefa T_i estará associada a um tempo médio de execução C_i, com o objetivo de obter uma visão mais realista do balanceamento de carga;
- Cada tarefa esporádica T_i estará associada a um intervalo médio entre ativações I_i, também com o objetivo de obter uma visão mais realista do balanceamento de carga. No caso das tarefas periódicas, temos $I_i = P_i$.

O valor E_b é uma medida do quanto a carga média do sistema está balanceada. Para cada processador H_j é calculada sua ocupação média U_j. A ocupação média de um processador é o somatório da ocupação média de todas as tarefas alocadas nele. A ocupação média de uma tarefa é dada por seu tempo de computação médio C_i dividido pela sua taxa de repetição média I_i. Como definido antes, a taxa de repetição média corresponde ao período das tarefas periódicas e ao intervalo médio entre ativações das tarefas esporádicas.

A energia E_b é obtida por: $E_b = \sum_{j=1}^{m} \left| U - U_j \right|$,

onde **m** é o número de processadores e **U** representa a taxa média de ocupação do sistema como um todo, dada por:

$$U = \frac{\sum_{i=1}^{n} \frac{C_i}{I_i}}{m} .$$

Uma solução de alocação onde todos os deadlines são garantidos possui E_e=0. Neste momento, o algoritmo de alocação vai buscar soluções com um valor E_b menor. Pela forma como E_b foi definido, isto significa buscar soluções onde todos os processadores apresentam, aproximadamente, a mesma utilização média. Desta forma, consegue-se evitar uma sobrecarga localizada em alguns processadores, o que tornaria improvável a execução precisa das tarefas ali alocadas. Assim, uma energia E_b menor favorece a execução de partes opcionais.

7.4 Experiências

Foram realizadas diversas experiências com a solução de alocação adotada. Os objetivos de tais experiências foi fixar os parâmetros do recozimento simulado para o

problema em questão e observar a vantagem ou desvantagem de usar balanceamento de carga como objetivo secundário. Em [TIN 92a] e [BUR 93] já foi analisada a eficiência do recozimento simulado em encontrar soluções para problemas de alocação de carga tempo real convencional (ou seja, composta por tarefas que não são do tipo impreciso). Esta análise não será repetida neste trabalho.

Para as experiências foram usadas cargas sintéticas compostas por 30 tarefas periódicas e 10 relações de precedência, executando em quatro processadores. As tarefas foram divididas em três grupos, com períodos variando entre 20 e 200, 200 e 2000 e finalmente entre 2000 e 20000. Períodos, deadlines, tempos de execução e *jitter* de liberação máximos foram escolhidos aleatoriamente. As relações de precedência foram escolhidas aleatoriamente. Elas formam atividades dentro da aplicação e exigem um ajuste nos períodos, deadlines, tempos de execução e *jitter* máximos sorteados antes.

O tempo máximo de execução da parte obrigatória de cada tarefa foi escolhido de tal sorte que a carga obrigatória total do sistema fosse controlada. Desta forma, as experiências realizadas sempre usaram 3 conjuntos de aplicações. Um conjunto possui carga obrigatória total correspondendo a 20% da capacidade do sistema. Os outros dois conjuntos possuem carga obrigatória total correspondendo a 50% e 80% da capacidade total de processamento do sistema.

Na implementação do recozimento simulado é necessária uma solução inicial. Para isto, cada tarefa é inicialmente alocada a um processador escolhido aleatoriamente. Soluções vizinhas são obtidas através da migração de uma tarefa para outro processador sorteado aleatoriamente (70% dos casos) ou pela troca dos processadores de duas tarefas quaisquer (30% dos casos).

Dentro da mecânica do recozimento simulado, toda solução vizinha escolhida deve ter sua energia computada. Para isto, é necessário recalcular os tempos de resposta de todas as tarefas da aplicação e compara-los com os respectivos deadline. Esta operação é demorada, sendo um dos principais fatores a determinar o tempo de execução do algoritmo de alocação. É possível incluir aqui uma otimização a nível de implementação que vai reduzir este tempo de execução. Uma vez que uma dada tarefa não é capaz de interferir com outras tarefas de prioridade superior (teorema 1 do capítulo 4), quando esta tarefa muda de processador, o tempo de resposta das tarefas com prioridade superior permanece o mesmo. Desta forma, para computar a energia de uma solução vizinha, é necessário apenas recalcular os tempos de resposta das tarefas com prioridade igual ou inferior às tarefas que mudaram de processador.

Para uma carga com o perfil descrito antes, foram usados os seguintes parâmetros:
$K_e = 10$;
$K_b = 1$;
$\alpha = 0.99$;
Temperatura inicial = 10 × número de processadores × número de tarefas;
Temperatura é reduzida sempre que uma das seguintes condições for satisfeita:

Número máximo de saltos =
(número de processadores × número de tarefas) / 2;

Número máximo de tentativas de salto =
número de processadores × número de tarefas;

Execução é concluida quando uma das seguintes condições for satisfeita:

Temperatura final = número de tarefas;

Número máximo de tentaivas sem salto =

$2 \times$ número de processadores $\times$ número de tarefas;

É importante notar que estes parâmetros foram escolhidos como o resultado de um compromisso entre a velocidade de execução do algoritmo e a qualidade final da solução alcançada. Para diferentes tipos de carga (densidade das relações de precedência, dificuldade de escalonamento, etc) é sempre possível otimizar estes parâmetros após algumas experiências.

Por exemplo, para determinadas aplicações, a temperatura inicial é muito alta. Uma temperatura inicial muito elevada não diminui a qualidade do resultado encontrado, mas aumenta o tempo de execução do algoritmo. Na implementação realizada do recozimento simulado, foi incluido um esquema de redução acelerada de temperatura. Sempre que a temperatura tivesse que ser reduzida devido ao número máximo de saltos ter ocorrido, era empregado um valor $\alpha = 0.9$, ao invés do valor 0.99 usual. O resultado desta medida é a redução rápida da temperatura quando ela for muito elevada, ou seja, quando toda tentativa de salto resultar efetivamente em um salto. Quando a temperatura baixa para valores onde nem todas tentativas de salto resultam efetivamente em salto, o valor usual de α volta a ser utilizado.

Dependendo da aplicação, a temperatura final pode ser atingida muito cedo, antes do "congelamento do sistema" no ponto de mínimo global. Neste caso, a última energia calculada poderá não ser a menor energia detectada ao longo da execução do algoritmo, mas um valor próximo daquele. Como um tratamento para este fenômeno, ao longo da execução do recozimento simulado é sempre mantida armazenada a alocação que resultou na menor energia observada. Desta forma, quando o algoritmo é concluido devido a temperatura final ter sido atingida, é possível considerar como solução a melhor alocação testada.

É importante notar a eficiência do recozimento simulado no que diz respeito ao objetivo secundário. Nas experiências realizadas foi observado que o balanceamento final da carga é muito melhor quando o objetivo secundário está presente. Foram realizadas experiências onde o mesmo conjunto de aplicações foi alocado segundo a função de energia adotada e também segundo uma função de energia que considera apenas o escalonamento das partes obrigatórias (fator E_e), ignorando o balanceamento de carga (fator E_b). Estas experiências resultaram em duas observações.

Em geral, existe uma realimentação positiva entre os fatores E_e e E_b. Uma melhor solução de escalonamento, na maioria das vezes, também leva a um melhor balanceamento da carga. Por sua vez, um melhor balanceamento de carga tente a facilitar o escalonamento do sistema. Esta realimentação positiva está presente na maioria das aplicações e faz com que soluções melhores sejam atingidas mais rapidamente. Entretanto, também existem alguns poucos casos onde a única solução possível para o escalonamento resulta em um péssimo balanceamento de carga.

Em termos numéricos, considerando o perfil de carga descrito antes, foi observado que a energia E_b da solução final é cerca de 10 vezes maior quando o recozimento simulado

ignora esta energia, comparado com quando o recozimento simulado inclui E_b no cálculo da energia do sistema. Considerando a ocupação dos processadores, o somatório dos desvios da ocupação média é 10 vezes menor quando o recozimento simulado inclui o objetivo secundário.

Como esperado, o melhor balanceamento de carga resulta em uma maior utilidade do sistema. Foram feitas simulações comparando o valor total obtido pela aplicação em tempo de execução quando o balanceamento de carga é ou não considerado na alocação. Em geral, o valor total obtido em tempo de execução é 5% maior quando o recozimento simulado possui, como objetivo secundário, o balanceamento da carga total.

A tabela 7.2 mostra o resultado obtido a partir de 20 aplicações aleatórias. Cada uma das aplicações é formada por 30 tarefas periódicas e inclui 10 relações de precedência. É suposta a existência de dependência intra-tarefa e inter-tarefa. Cada aplicação possui uma carga obrigatória total de 50% e uma carga opcional total de 120%, considerando que é executada em um sistema distribuído composto por 4 processadores. Cada uma destas 20 aplicações foi alocada duas vezes. Na primeira vez, a função Energia usada pelo recozimento simulado considerou apenas a escalonabilidade do sistema, ignorando completamente o aspecto balanceamento. Em outras palavras, foi usado $E = K_e \times E_e$. Na segunda vez que cada aplicação alocada o recozimento simulado usou a função de energia descrita na seção 7.3.2.

Para cada uma das duas alocações de cada aplicação foram simuladas todas as políticas de admissão apresentadas no capítulo anterior. A coluna "$E = K_e . E_e$" mostra o valor total obtido em tempo de execução quando a alocação ignora o aspecto balanceamento de carga. A coluna "$E = K_e . E_e + K_b . E_b$" mostra o valor total obtido exatamente pelas mesmas aplicações, quando a alocação considera o balanceamento de carga. Os valores apresentados são relativos ao valor que a política FCFS apresentou na alocação sem objetivo secundário. Por exemplo, o valor 1.059 na segunda coluna da linha correspondente ao FCFS indica que, quando o objetivo secundário está presente, esta política é capaz de obter 5.9% a mais de valor da mesma aplicação.

	$E = K_e . E_e$	**$E = K_e . E_e + K_b . E_b$**
FCFS	1.000	1.059
AVDT	1.288	1.337
CVDT	1.289	1.347
INTER	1.292	1.355

Tabela 7.2 - Valores obtidos pelas diferentes políticas de admissão.

7.5 Conclusões

Neste capítulo foi discutido o problema da alocação de tarefas imprecisas em ambiente distribuído. Após uma revisão da literatura, foi escolhido o método do recozimento simulado. A escolha é justificada pela complexidade do problema em questão, o que dificulta a escolha de boas heurísticas.

A aplicação do recozimento simulado exige a definição de uma função de energia capaz de capturar as diferentes alternativas ("tradeoffs") possíveis na construção de uma solução. No caso do problema abordado, a função energia considera três fatores. Primeiro, o fato de tarefas deverem ter o seu deadline garantido, quando apenas as partes obrigatórias são consideradas. Segundo, o fato da carga média do sistema estar ou não balanceada, de modo a aumentar as chances das partes opcionais. Finalmente, as restrições de alocação impostas pelo projeto do sistema.

As experiências realizadas confirmaram que o recozimento simulado é uma abordagem viável para o problema da alocação de tarefas com restrições de tempo real em sistemas distribuídos. Além disto, elas mostraram que a inclusão do balanceamento de carga na função de energia não somente é viável, mas também resulta em sistemas com a carga muito melhor balanceada, mostrando a eficiência do método. Simulações mostraram que, ao incluir o balanceamento de carga nos objetivos da alocação, o sistema é capaz de gerar mais valor em tempo de execução. Isto é válido para todas as políticas de admissão discutidas no capítulo 6.

8 Conclusões

8.1 Revisão dos Objetivos

O objetivo geral desta tese é mostrar como aplicações de tempo real, construídas a partir do conceito de Computação Imprecisa, podem ser escalonadas em ambiente distribuído. Este problema geral de escalonamento pode ser dividido em quatro problemas específicos. São eles:

- Como garantir que as partes obrigatórias das tarefas serão concluídas antes dos respectivos deadlines, em um ambiente onde tarefas podem executar em diferentes processadores e o emprego de mensagens gera atrasos e cria relações de precedência entre as tarefas.

- Como determinar que a execução de uma parte opcional não irá comprometer a execução das partes obrigatórias, previamente garantidas, fazendo com que uma ou mais tarefas percam o seu deadline.

- Quais critérios utilizar para escolher quais partes opcionais devem ser executadas, na medida em que o recurso "tempo de processador disponível" não permite a execução de todas as partes opcionais da aplicação.

- Como resolver qual tarefa executa em qual processador, de forma que todas as partes obrigatórias das tarefas possam ser garantidas e que as partes opcionais estejam distribuídas de forma que sua chance de execução seja maximizada.

Ao resolver estas quatro questões, estamos mostrando que efetivamente Computação Imprecisa pode ser usada como base para a construção de aplicações distribuídas de tempo real.

8.2 Resumo do Trabalho Realizado

Nesta seção é feita uma revisão do trabalho realizado e de como este trabalho atacou os problemas listados na seção anterior. O capítulo 1 da tese descreveu o contexto onde este trabalho está inserido e seus objetivos. O capítulo 2 fez uma revisão da literatura de escalonamento tempo real e, em particular, do escalonamento tempo real baseado em Computação Imprecisa.

O capítulo 3 descreveu a abordagem geral empregada neste trabalho. Foram discutidos os principais problemas envolvidos. Os quatro objetivos específicos, listados na seção anterior, surgiram de uma discussão da problemática relacionada com o escalonamento de tarefas imprecisas em ambiente distribuído, apresentada no capítulo 3. Cada um destes quatro objetivos específicos deve ser alcançado através de um algoritmo próprio. A figura 3.1 mostrou como estes quatro algoritmos estão relacionados entre si.

O capítulo 3 também descreveu o modelo básico de tarefas adotado neste trabalho. Este modelo inclui tarefas periódicas e esporádicas, ambiente distribuído, relações de precedência, liberação dinâmica, *jitter* na liberação e deadline menor ou igual ao período.

Tarefas possuem parte obrigatória e opcional, prioridades fixas e podem ser preteridas ("preemptive scheduler") a qualquer instante por uma tarefa de prioridade superior. Além disto, o modelo básico de tarefas inova ao permitir que tarefas imprecisas possuam dependências tipo intra-tarefa e tipo inter-tarefa, sem com isto alterar a carga obrigatória do sistema. Dependência intra-tarefa decorre da importância de ocasionalmente executar uma dada tarefa de maneira precisa. Dependência inter-tarefa surge quando a qualidade dos dados de entrada de uma dada tarefa afeta o seu tempo máximo de execução.

No capítulo 4 foi desenvolvido um teste de escalonabilidade para as partes obrigatórias das tarefas. O objetivo do teste é mostrar que todas as tarefas da aplicação atendem ao seu respectivo deadline quando partes opcionais não são executadas. No teste desenvolvido, o sistema de tarefas original é transformado em um sistema equivalente sem relações de precedência, o qual pode ser analisado através de métodos presentes na literatura para tarefas independentes. Um total de nove regras de transformação foram enunciadas e demonstradas na forma de teoremas. Elas serviram de base para a construção de um algoritmo capaz de realizar a transformação citada. No final do capítulo um exemplo numérico ilustrou a vantagem do algoritmo proposto em comparação com métodos que simplesmente transformam relações de precedência em *jitter* de liberação. O emprego do algoritmo proposto neste trabalho resultou, no exemplo, em tempos máximos de resposta que são mais próximos do valor real e, portanto, menos pessimistas.

É importante notar que o teste de escalonabilidade desenvolvido no capítulo 4 pode ser também empregado na análise da escalonabilidade de tarefas normais, isto é, tarefas compostas somente por partes obrigatórias. Neste caso, a solução para o problema do escalonamento como um todo se resume às etapas de alocação e garantia das partes obrigatórias em tempo de projeto.

O capítulo 5 discutiu como as folgas existentes no sistema podem ser utilizadas para a execução de partes opcionais. Isto deve ser feito sem que as partes obrigatórias sejam prejudicadas. O capítulo 5 inicia descrevendo diversas abordagens presentes na literatura: baseadas em planejamento, baseadas em servidores, tomada estática de folga e tomada dinâmica de folga. Neste trabalho foi adotada a abordagem da tomada dinâmica de folga aproximada (DASS).

O capítulo 5 mostrou como o DASS pode ser adaptado para o contexto deste trabalho. Tarefas esporádicas com *jitter* de liberação recebem tratamento especial no momento do cálculo da folga. Tarefas com sucessores remotos precisam preservar o seu tempo máximo de resposta, para evitar que o atraso propagado através das relações de precedência faça alguma tarefa remota perder o seu deadline. A existência de dependência inter-tarefa foi aproveitada para aumentar a eficiência da detecção de folgas.

A quantidade de memória necessária para a implementação do algoritmo DASS modificado não é preocupante, pois as tabelas necessárias em cada processador são proporcionais ao número de tarefas alocadas ao respectivo processador. O capítulo 5 inclui também um exemplo numérico, que ilustra a operação do algoritmo DASS modificado. O capítulo 5 termina com uma seção contendo considerações gerais sobre o algoritmo proposto.

O capítulo 6 abordou a questão da escolha de quais partes opcionais devem utilizar os recursos disponíveis em um dado momento. Foram descritas duas heurísticas originais

(CVDT e INTER) para serem usadas como políticas de admissão. As heurísticas descritas procuram tirar proveito das dependências intra-tarefa e inter-tarefa existentes no sistema. O algoritmo CVDT compara a densidade de valor da parte opcional em questão com a densidade média obtida pelo sistema, corrigida em função da carga no processador. A parte opcional é admitida quando sua densidade de valor for superior a média corrigida do sistema. O algoritmo INTER está baseado no mesmo princípio do CVDT, mas ele considera também os ganhos futuros de valor gerados por uma execução precisa de tarefa. Tais ganhos são decorrentes das dependências inter-tarefa.

O comportamento das heurísticas propostas foi analisado no capítulo 6 através de simulação. Seu comportamento foi comparado ao de dois outros algoritmos presentes na literatura. As simulações mostraram que, em determinadas situações, as heurísticas propostas neste trabalho apresentam uma eficiência superior às existentes na literatura, no que diz respeito a maximização da utilidade do sistema.

Finalmente, o capítulo 7 abordou a questão da alocação de tarefas imprecisas. Foi usado o método do recozimento simulado para fazer a alocação. O objetivo primário da alocação é obter uma garantia para as partes obrigatórias de todas as tarefas. Para tanto, o teste de escalonabilidade do capítulo 4 é usado. O processo de alocação tem como objetivo secundário o balanceamento de carga, de forma a aumentar a chance das partes opcionais serem executadas.

Foram realizadas diversas experiências com aplicações geradas aleatoriamente, dentro de determinados padrões de utilização. As experiências realizadas mostraram que a inclusão do balanceamento de carga na função de energia do recozimento simulado resulta em sistemas com a carga muito melhor balanceada. Considerando a utilização dos processadores, o somatório dos desvios da utilização média é 10 vezes menor quando o recozimento simulado inclui o objetivo secundário, o que mostra a eficiência do método.

Simulações mostraram também que, ao incluir o balanceamento de carga nos objetivos da alocação, o sistema é capaz de gerar mais valor em tempo de execução. O capítulo 7 descreveu simulações comparando o valor total obtido em tempo de execução quando o balanceamento de carga é ou não considerado na alocação. Em geral, o valor total obtido em tempo de execução é 5% maior quando o balanceamento da carga é incluído no recozimento simulado como objetivo secundário. Este ganho de 5% no valor total do sistema foi observado em todas as quatro políticas de admissão discutidas no capítulo 6.

Pelo descrito acima, pode-se observar que o problema do escalonamento de tarefas imprecisas em ambiente tempo-real distribuído foi discutido, como um todo, no capítulo 3. Após esta discussão geral, os quatro problemas específicos citados na seção 8.1 foram abordados individualmente nos capítulos 4 a 7.

As simulações realizadas no capítulo 6 proporcionaram uma oportunidade para empregar a solução completa proposta neste trabalho. Inicialmente, aplicações foram geradas aleatoriamente, dentro de determinados padrões de utilização obrigatória e opcional. Em seguida, as tarefas eram alocadas conforme a solução baseada em recozimento simulado proposta pelo capítulo 7. Como parte do problema de alocação, a escalonabilidade das tarefas era testada através do algoritmo proposto pelo capítulo 4. Uma vez feita a alocação, era simulada a execução da aplicação, com o objetivo de determinar o valor total obtido. Durante a simulação da execução foi usada a solução proposta pelo

capítulo 5 para identificar folgas no sistema e, com base nelas, estabelecer um teste de aceitação. Finalmente, as políticas de admissão propostas no capítulo 6 foram simuladas e o respectivo valor obtido foi computado.

Com base no que foi exposto nesta seção, é possível afirmar que o objetivo da tese foi alcançado. O texto apresenta uma solução completa para o problema do escalonamento de tarefas imprecisas em ambiente tempo real distribuído.

8.3 Contribuições ao Estado da Arte

De uma forma geral, esta tese fornece uma solução completa para o escalonamento de tarefas imprecisas em sistemas tempo real distribuídos. Não é do conhecimento do autor nenhum outro trabalho na literatura que aborde todos os aspectos deste problema, como foi feito aqui. De uma forma particular, pode-se citar:

- Definição de um modelo de tarefas que contempla as dependências inter-tarefa e intra-tarefa existentes em sistemas de tarefas imprecisas, ao mesmo tempo que mantém intacto o caráter obrigatório e opcional das partes de uma tarefa.

- Um método original para realizar o teste de escalonabilidade em sistemas de tarefas executados em ambiente distribuído, quando as tarefas apresentam relações de precedência e liberação dinâmica.

- Uma adaptação original do algoritmo de tomada aproximada de folga para sistemas que incluem execução em ambiente distribuído, tarefas esporádicas, relações de precedência e dependência inter-tarefa.

- Heurísticas originais para serem utilizadas como políticas de admissão em sistemas de tarefas imprecisas.

- Um método de alocação para sistemas de tarefas imprecisas que contempla o aspecto da distribuição da carga opcional, além da usual garantia para partes obrigatórias.

8.4 Temas para Futuras Pesquisas

Na medida em que, cada vez mais, aplicações tempo real envolvem aspectos críticos juntamente com diversos aspectos não críticos, é opinião do autor que a técnica de Computação Imprecisa será um caminho atraente para a construção de sistemas tempo real. Isto é válido tanto para sistemas distribuídos como para processamento paralelo e processamento centralizado.

A partir deste trabalho pode-se imaginar diversos temas de pesquisa. Abaixo estão listados os mais importantes.

- Investigação de soluções para o problema da alocação baseadas em algoritmos genéticos.

- Investigação de outros objetivos secundários para o processo de alocação, além do simples balanceamento de carga, que resultem na geração de maior valor em tempo de execução.

- Investigação de métodos para a definição dos valores nominais das tarefas em tempo de projeto, a partir da especificação do sistema.

- Incorporação da técnica de Computação Imprecisa às técnicas de construção de software existentes hoje. Em particular, técnicas orientadas a objeto.

- Desenvolvimento de linguagens de programação apropriadas para a Computação Imprecisa, considerando-se suas diferentes formas de programação.

- Investigação de algoritmos para teste de aceitação que apresentem um custo computacional menor do que a tomada de folga.

- Investigação do impacto causado por mudanças no modo de operação ("mode change") sobre as políticas de admissão discutidas no capítulo 6.

O principal obstáculo hoje à utilização da técnica de Computação Imprecisa em aplicações de tempo real é o fato das técnicas tradicionais de engenharia de software não incorporarem o conceito de execução imprecisa. Neste sentido, existe muito trabalho a ser feito na área de Engenharia de Software. Uma das poucas ferramentas existentes hoje que suporta este conceito é o PERTS, descrito em [LIU 93]. Como colocado em [AUD 91c], "o uso disseminado de técnicas, tais como Computação Imprecisa, somente acontecerá se elas forem integradas aos métodos usuais de engenharia de software".

9 Bibliografia

[AUD 90a] N. C. Audsley. Deadline Monotonic Scheduling. Department of Computer Science report YCS 146, University of York, october 1990.

[AUD 90b] N. Audsley, A. Bhattacharyya, A. Burns, G. Fohler, H. Kantz, H. Kopetz, J. A. McDermid, W. Schütz, R. Zainlinger. Timeliness - Sumary and Conclusions. Esprit Bra Project 3092 "Predictably Dependable Computing Systems", First Year Report Task B, Vol. 2, 1990.

[AUD 90c] N. Audsley, A. Burns. Timeliness Part II - Real-Time System Scheduling. Esprit Bra Project 3092 "Predictably Dependable Computing Systems", First Year Report Task B, Vol. 2, 1990.

[AUD 91a] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, may 1991.

[AUD 91b] N. C. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times. Department of Computer Science report YCS 164, University of York, 1991.

[AUD 91c] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings. Incorporating Unbounded Algorithms into Predictable Real-Time Systems. Technical Report RTRG/91/102. University of York, Department of Computer Science, september 1991.

[AUD 93a] N. C. Audsley, A. Burns, A. J. Wellings. Deadline Monotonic Scheduling Theory and Application. Control Engineering Practice, Vol. 1, No. 1, pp. 71-78, february 1993.

[AUD 93b] N. Audsley, K. Tindell, A. Burns. The End of the Line for Static Cyclic Scheduling? Proceedings of the Fifth Euromicro Workshop on Real-Time Systems, IEEE Computer Society Press, pp. 36-41, june 1993.

[AUD 93c] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Software Engineering Journal, Vol. 8, No. 5, pp.284-292, 1993.

[AUD 93d] N. C. Audsley. Flexible Scheduling of Hard Real-Time Systems. Department of Computer Science Thesis, University of York, 1994.

[AUD 94a] N. C. Audsley, A. Burns, R. I. Davis, A. J. Wellings. Integrating Best Effort and Fixed Priority Scheduling. Proceedings of the IEEE Real-Time Systems Symposium, pp. 12-21, San Juan, Puerto Rico, december 1994.

[AUD 94b] N. C. Audsley, R. I. Davis, A. Burns. Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 12-21, San Juan, Puerto Rico, december 1994.

[BAK 91] T. P. Baker. Stack-Based Scheduling of Realtime Processes. The Journal of Real-Time Systems, Vol. 3, pp. 67-90, 1991.

[BET 92] R. Bettati, J. W.-S. Liu. End-to-End Scheduling to Meet Deadlines in Distributed Systems. Proceedings of the 12th International Conference on Distributed Computing Systems, pp. 452-459, june 1992.

[BUR 91] A. Burns, A. J. Wellings. Criticality and Utility in the Next Generation. The Journal of Real-Time Systems, Vol. 3, correspondence, pp. 351-354, 1991.

[BUR 93] A. Burns, M. Nicholson, K. Tindell, N. Zhang. Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System. Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, pp. 11-20, 1993.

[BUR 95] A. Burchard, J. Liebeherr, Y. Oh, S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. IEEE Transactions on Computers, Vol.44, No.12, dec 1995, pp.1429-1442.

[BUR 96] A. Burns, A. J. Wellings. Dual Priority Scheduling in ADA 95 and Real-Time Posix. Proceedings of the 21st IFAC/IFIP Workshop on Real Time Programming, pp. 45-50, Gramado-RS-Brazil, november 1996.

[CHA 95] S. Chatterjee, J. Strosnider. Distributed Pipeline Scheduling: End-to-End Analysis of Heterogeneous, Multi-Resource Real-Time Systems. Proceedings of the 15th International Conference on Distributed Computing Systems, may 1995.

[CHE 88] S. Cheng, J. A. Stankovic, K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey. In Hard Real-Time Systems: Tutorial, ed. J. A. Stankovic e K. Ramamritham, pp. 150-173, IEEE Computer Society Press, 1988.

[CHE 90a] H. Chetto, M. Silly, T. Bouchentouf. Dynamic Scheduling of Real-Time Tasks under Precedence Constraints. The Journal of Real-Time Systems, Vol. 2, pp. 181-194, 1990.

[CHE 90b] M.-I. Chen, K.-J. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. Real-Time Systems, Vol. 2, pp. 325-346, 1990.

[CHU 90] J. Y. Chung, J. W. S. Liu, K. J. Lin. Scheduling Periodic Jobs that Allow Imprecise Results. IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1156-1174, september 1990.

[DAM 89] A. Damm, J. Reisinger, W. Schwabl, H. Kopetz. The Real-Time Operating System of Mars. ACM Operating Systems Review, Vol. 23, No. 3, pp. 141-151, 1989.

[DAV 93a] R. I. Davis, K. W. Tindell, A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 222-231, 1993.

[DAV 93b] R. I. Davis. Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems. Department of Computer Science report YCS, University of York, 1993.

[DAV 95] R. Davis, S. Punnekkat, N. Audsley, A. Burns. Flexible Scheduling for Adaptable Real-Time Systems. Proceedings of the IEEE Real-Time Technology and Applications Symposium, pp. 230-239, may 1995.

[ELH 94] C. McElhone. Adapting and Evaluating Algorithms for Dynamic Schedulability Testing. Department of Computer Science report YCS, University of York, february 1994.

[FEN 94] W. Feng, J. W.-S. Liu. Algorithms for Scheduling Tasks with Input Error and End-to-End Deadlines. University of Illinois at Urbana-Champaign, Department of Computer Science, Technical Report #1888, 1994.

[FEN 96] W. Feng, J. W.-S. Liu. Performance of a Congestion Control Scheme on an ATM Switch. Proceedings of the International Conference on Networks, Orlando, Florida, january 1996.

[GAR 94] A. Garvey, V. Lesser. A Survey of Research in Deliberative Real-Time Artificial Intelligence. Real-Time Systems, Vol. 6, pp. 317-347, 1994.

[GER 94] R. Gerber, S. Hong, M. Saksena. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. Proceedings of the IEEE Real-Time Systems Symposium, december 1994.

[HAR 94] M. G. Harbour, M. H. Klein, J. P. Lehoczky. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. IEEE Transactions on Software Engineering, Vol. 20, No. 1, pp. 13-28, january 1994.

[HO 92a] K. I.-J. Ho, J. Y.-T. Leung, W.-D. Wei. Scheduling Imprecise Computation Tasks with 0/1-Constraint. Technical Report, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 1992.

[HO 92b] K. I.-J. Ho, J. Y.-T. Leung, W.-D. Wei. Minimizing Constrained Maximum Weighted Error for Doubly Weighted Tasks. Technical Report, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 1992.

[HO 94] K. I.-J. Ho, J. Y.-T. Leung, W.-D. Wei. Minimizing Maximum Weighted Error for Imprecise Computation Tasks. Journal of Algorithms, 16, pp. 431-452, 1994.

[HOU 92a] C.-J. Hou, K. G. Shin. Allocation of Periodic Task Modules with Precedence and Deadlines Constraints in Distributed Real-Time Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 146-155, 1992.

[HOU 92b] E. S. H. Hou, H. Ren, N. Ansari. Efficient Multiprocessor Scheduling Based on Genetic Algorithms. In "Dynamic, Genetic, and Chaotic Programming", Ed. B. Soucek, John Wiley & Sons, 1992.

[HUA 95] X. Huang, A. M. K. Cheng. Applying Imprecise Algorithms to Real-Time Image and Video Transmission. Proceedings of the IEEE Workshop on Real-Time Applications. Chicago, may 1995.

[JEF 92] K. Jeffay. Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 89-99, 1992.

[JEF 93] K. Jeffay, D. L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 212-221, december 1993.

[JEN 85] E. D. Jensen, C. D. Locke, H. Tokuda . A Time-Driven Scheduling Model for Real-Time Operating Systems. Proceedings of the Real-Time Systems Symposium, pp.112-122, december 1985.

[KEN 91] K. B. Kenny, K.-J. Lin. Building Flexible Real-Time Systems Using the Flex Language. IEEE Computer, pp.70-78, may 1991.

[KIR 83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. Optimization by Simulated Annealing. Science (220), pp. 671-680, 1983.

[KLE 94] M. H. Klein, J. P. Lehoczky, R. Rajkumar. Rate-Monotonic Analysis for Real-Time Industrial Computing. IEEE Computer, pp. 24-33, january 1994.

[KOP 89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabi, C. Senft, R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. IEEE Micro, pp. 25-40, february 1989.

[KOP 92] H. Kopetz. Scheduling. An Advanced Course on Distributed Systems, Estoril-Portugal, 1992.

[KOR 92] G. Koren, D. Shasha. Dover: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 290-299, 1992.

[KRO 94] E. Krotkov, R. Hoffman. Terrain Mapping for a Walking Planetary Rover. IEEE Transactions on Robotics and Automation, Vol. 10, No. 6, pp. 728-739, december 1994.

[LAN 90] G. L. Lann. Critical Issues for the Development of Distributed Real-Time Computing Systems. INRIA - Institut National de Recherche en Informatique et en Automatique, Rapports de Recherche N. 1274, août 1990.

[LEH 87] J. P. Lehoczky, L. Sha, J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard-Real-Time Environments. Proceedings of the IEEE Real-Time Systems Symposium, San Jose-CA, pp.261-270, 1987.

[LEH 89] J. P. Lehoczky, L. Sha, Y. Ding. The Rate Monotonic Scheduling Algorithm - Exact Characterization and Average-Case Behavior. Proceedings of the IEEE Real-Time Systems Symposium, december 1989.

[LEH 90] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. Proceedings of the 11th Real-Time Systems Symposium, pp. 201-209, december 1990.

[LEH 92] J. P. Lehoczky, S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 110-123, 1992.

[LEU 82] J. Y. T. Leung, J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. Performance Evaluation, 2 (4), pp. 237-250, december 1982.

[LIN 87a] K. J. Lin, S. Natarajan, J. W. S. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In Proc. IEEE 8th Real-Time Systems Symposium, San Jose, California, USA, December 1987.

[LIN 87b] K. J. Lin, S. Natarajan, J. W. S. Liu. Scheduling Real-Time, Periodic Jobs Using Imprecise Results. In Proc. IEEE 8th Real-Time Systems Symposium, San Jose, California, USA, December 1987.

[LIU 73] C. L. Liu, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, Vol. 20, No. 1, pp. 46-61, january 1973.

[LIU 88] J. W. S. Liu, K.-J. Lin. On Means to Provide Flexibility in Scheduling. Ada Letters, 1988 Special Edition, Vol. VIII, No. 7, pp. 32-34.

[LIU 91] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung, W. Zhao. Algorithms for Scheduling Imprecise Computations. IEEE Computer, may 1991, pp 58-68.

[LIU 93] J. W. S. Liu, J.-L. Redondo, Z. Deng, T.-S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, W.-K. Shih. PERTS: A Prototyping Environment for Real-Time Systems. Technical Report, Department of Computer Science, University of Illinois, may 1993.

[LIU 94] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung. Imprecise Computations. Proceedings of the IEEE, Vol. 82, No. 1, pp. 83-94, january 1994.

[LOC 92] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. The Journal of Real-Time Systems, Vol. 4, pp. 37-53, 1992.

[MAG 92] M. F. Magalhães. Tópicos Especiais em Escalonamento de Sistemas em Tempo Real - Notas de Aula. Laboratório de Controle e Microinformática, UFSC, Relatório Técnico RT92/02, abril 1992.

[MOT 93] L. Motus. Time Concepts in Real-Time Software. Control Engineering Practice, Vol. 1, No. 1, pp. 21-33, february 1993.

[OLI 95] R. S. Oliveira, J. S. Fraga. Um Modelo de Tarefas Distribuído Empregando Computação Imprecisa. Anais do XXII Seminário Integrado de Software e Hardware, pp. 693-704, Gramado-RS, agosto 1995.

[OLI 96a] R. S. Oliveira, J. S. Fraga. Uma Solução Mista para o Escalonamento Baseado em Prioridades de Aplicações Tempo Real Críticas. Anais do XXIII Seminário Integrado de Software e Hardware, pp. 143-153, Recife-PE, agosto 1996.

[OLI 96b] R. S. Oliveira, J. S. Fraga. Scheduling Imprecise Computation Tasks with Intra-Task / Inter-Task Dependence. Proceedings of the 21st IFAC/IFIP Workshop on Real Time Programming, Gramado-RS-Brasil, pp. 51-56, november 1996.

[PAR 82a] R. G. Parker, R. L. Rardin. An Overview of Complexity Theory in Discrete Optimizations: Part I - Concepts. IIE Transactions, Vol. 14, No. 1, pp. 3-10, march 1982.

[PAR 82b] R. G. Parker, R. L. Rardin. An Overview of Complexity Theory in Discrete Optimization: Part II - Results and Implications. IIE Transactions, Vol. 14, No. 2, pp. 83-89, june 1982.

[RAM 89] K. Ramamritham, J. A. Stankovic, W. Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. IEEE Transactions on Computers, Vol. 38, No. 8, pp. 1110-1123, august 1989.

[RAM 90] K. Ramamritham, J. A. Stankovic, P.-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, pp. 184-194, april 1990.

[RAM 94] K. Ramamritham, J. A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. Proceedings of the IEEE, Vol. 82, No. 1, pp. 55-67, january 1994.

[REM 93] U. Rembold, B. O. Nnaji, A. Storr. Computer Integrated Manufacturing and Engineering. Addison-Wesley Publishing Company, ISBN 0-201-56541-2, 1993.

[SCH 92] K. Schwan, H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. IEEE Transactions on Software Engineering, Vol. 18, No. 8, pp. 736-748, august 1992.

[SHA 90] L. Sha, R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An approach to Real-Time Synchronization. IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175-1185, september 1990.

[SHA 93] L. Sha, S. S. Sathaye. A Systematic Approach to Designing Distributed Real-Time Systems. IEEE Computer, pp. 68-78, september 1993.

[SHA 94] L. Sha, R. Rajkumar, S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. Proceedings of the IEEE, Vol. 82, No. 1, pp. 68-82, january 1994.

[SHI 89] W.-K. Shih, J. W. S. Liu, J.-Y. Chung, D. W. Gillies. Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error. ACM Operating Systems Review, pp. 14-28, july 1989.

[SHI 92] W.-K. Shih, J. W. S. Liu. On-line Scheduling of Imprecise Computations to Minimize Error. Proceedings of IEEE Real-Time Systems Symposium, pp. 280-289, 1992.

[SHI 94] K. G. Shin, P. Ramanathan. Real-Time Computing: A New Discipline of Computer Science and Engineering. Proceedings of the IEEE, Vol. 82, No. 1, pp. 6-24, january 1994.

[SPR 89] B. Sprunt, L. Sha, J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. The Journal of Real-Time Systems, Vol. 1, pp. 27-60, 1989.

[STA 88] J. A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. IEEE Computer, pp. 10-19, october 1988.

[SUN 94] J. Sun, R. Bettati, J. W.-S. Liu. An End-to-End Approach to Scheduling Periodic Tasks with Shared Resources in Multiprocessor Systems. Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software, pp. 18-22, 1994.

[SUN 95] J. Sun, J. W.-S. Liu. Bounding the End-to-End Response Time in Multiprocessor Real-Time Systems. Proceedings of Workshop on Parallel and Distributed Real-Time Systems, pp.91-98, Santa Barbara, CA, april 1995

[SUN 96a] J. Sun, J. W.-S. Liu. Synchronization Protocols in Distributed Real-Time Systems. Proceedings of 16th International Conference on Distributed Computing Systems, may 1996.

[SUN 96b] J. Sun, J. W.-S. Liu. Bounding the End-to-End Response Times of Tasks in a Distributed Real-Time System Using the Direct Synchronization Protocol. Technical Report UIUCDCS-R-96-1949, University of Illinois at Urbana-Champaign, Department of Computer Science, june 1996.

[THU 93] S. Ramos-Thuel, J. P. Lehoczky. On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 160-171, 1993.

[THU 94] S. R. Ramos-Thuel, J. P. Lehoczky. Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems Using Slack Stealing. Proceedings of the 15th IEEE Real-Time Systems Symposium, pp.22-33, December 1994.

[TIA 94] T.-S. Tia, J. W. S. Liu, M. Shankar. Aperiodic Request Scheduling in Fixed-Priority Preemptive Systems. Department of Computer Science Technical Report #1859, University of Illinois at Urbana-Champaign, 1994.

[TIN 92a] K. W. Tindell, A. Burns, A. J. Wellings. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. Real-Time Systems, Vol. 4, No. 2, pp. 145-165, june 1992.

[TIN 94a] K. W. Tindell, A. Burns, A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. Real-Time Systems, pp. 133-151, 1994.

[TIN 94b] K. W. Tindell. Fixed Priority Scheduling of Hard Real-Time Systems. Department of Computer Science thesis, University of York, 1994.

[TIN 94c] K. Tindell, J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. Microprocessors and Microprogramming, 40, 1994.

[VER 91] J. P. C. Verhoosel, E. J. Luit, D. K. Hammer, E. Jensen. A Static Scheduling Algorithm for Distributed Hard Real-Time Systems. The Journal of Real-Time Systems, Vol. 3, pp. 227-240, 1991.

[XU 93a] J. Xu, D. L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. IEEE Transactions on Software Engineering, Vol. 19, No. 1, pp. 70-84, January 1993.

[YU 92] A. C. Yu, K.-J. Lin. A Scheduling Algorithm for Replicated Real-Time Tasks. IEEE IPCCC'92, pp. 395-402, 1992.

[ZHA 87] W. Zhao, K. Ramamritham, J. A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. IEEE Transactions on Computers, Vol. C-36, No. 8, pp. 949-960, august 1987.