UNIVERSIDADE FEDERAL DE SANTA CATARINA

Programa de Pós-Graduação em Engenharia Elétrica

Área de Concentração: Sistemas de Informação

O Emprego da Computação Imprecisa em Sistemas de Tempo Real Distribuídos

Monografia submetida ao exame de qualificação para o
doutorado em Engenharia Elétrica

Doutorando: Rômulo Silva de Oliveira

Orientador: Prof. Dr. Joni da Silva Fraga

Florianópolis, dezembro de 1994

O Emprego da Computação Imprecisa em Sistemas de Tempo Real Distribuídos

Rômulo Silva de Oliveira
Laboratório de Controle e Microinformática
EEL/CTC/UFSC

e-mail: romulo@lcmi.ufsc.br

06 de dezembro de 1994

Resumo

Neste trabalho estuda-se a utilização da técnica Computação Imprecisa em sistemas tempo real distribuídos. Inicialmente, os conceitos básicos de tempo real são definidos. As variações existentes nos modelos de tarefas adotados são discutidas. As principais abordagens para o problema de escalonamento tempo real são apresentadas e analisadas. Estas mesmas abordagens são exemplificadas e então comparadas. A seguir, a técnica Computação Imprecisa é descrita. São apresentadas motivações para seu emprego, formas de programação e as variações encontradas com respeito ao objetivo dos algorítmos de escalonamento. São ainda apresentados algorítmos de escalonamento que incorporam os conceitos de Computação Imprecisa e feitas considerações a respeito do estado atual da pesquisa nesta área. Finalmente, são discutidas questões ligadas à Computação Imprecisa que permanecem em aberto na literatura. Estas questões servem de ponto de partida para a proposição do trabalho a ser desenvolvido. É apresentado um plano de trabalho para o restante do curso.

Abstract

In this work, it is studied the utilization of the Imprecise Computation technique in distributed real-time systems. First, the basic concepts of real-time are defiined. The variations usually found in the adopted task models are discussed. The most important approaches to real-time scheduling are presented and analised. These same approaches are exemplified and than compared. After, the Imprecise Computation technique is described. Motivations for its use are presented, followed by the presentation of programming methods and variations with respect to the goal of scheduling algorithms. Scheduling algorithms that incorporate the Imprecise Computation concepts are also showed and the state of the art discussed. At last, questions related to Imprecise Computation that are still opened in the literature are discussed. Those questions are used as a starting point for the proposition of the work to be undertook. A work plan for the remaining of the course is presented.

Conteúdo

Lista de Figuras

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 à 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.

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.

Um problema básico encontrado na construção de sistemas de tempo real é a alocação e o escalonamento das tarefas nos recursos computacionais disponíveis. Existe uma dificuldade intrinsica entre 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.

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.

1.2 Objetivos da Proposta de Tese

O objetivo geral desta proposta de tese é analisar as alternativas existentes para solucionar o problema de escalonamento tempo real em ambiente distribuído, visando propor uma solução para o problema. Esta solução terá como base a técnica de Computação Imprecisa aplicada ao contexto em questão.

Especificamente, este trabalho pretende:

- Descrever e analisar as abordagens usuais para o problema de escalonamento tempo real em geral, discutindo suas vantagens e desvantagens;

- Descrever e analisar as propostas existentes na literatura que empregam Computação Imprecisa no escalonamento tempo real, identificando os problemas e limitações que existem hoje no emprego desta técnica;

- Sugerir formas para o emprego de Computação Imprecisa em sistemas tempo real distribuídos, particularmente em aplicações típicas do ambiente que interessa ao LCMI (automação industrial).

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, algorítmos, 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

Este trabalho está dividido em 4 capítulos. O capitulo 1 descreveu o contexto geral no qual o trabalho está inserido. Também foram rapidamente apresentados os objetivos desta proposta.

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. Estas mesmas abordagens são exemplificadas através da apresentação de algorítmos de escalonamento importantes, dentro de cada abordagem. No final do capítulo são discutidas as vantagens e desvantagens de cada abordagem.

O capítulo 3 trata especificamente de Computação Imprecisa. Inicialmente, esta técnica é descrita. São apresentadas motivações para seu emprego, formas de programação e as variações encontradas na literatura com respeito ao objetivo dos algorítmos de escalonamento baseados nesta técnica. Também são apresentados algorítmos de escalonamento apropriados para modelos de tarefas que incorporam os conceitos de Computação Imprecisa. No final do capítulo são feitas algumas considerações a respeito do estado atual da pesquisa nesta área.

Finalmente, o capítulo 4 contém a proposta de tese propriamente dita. O capítulo inicia listando algumas questões ligadas à Computação Imprecisa que permanecem ainda em aberto na literatura e servem de ponto de partida para o trabalho proposto. A descrição da proposta procura mostrar o problema a ser atacado, seu contexto e sua relevância. O

capítulo termina com uma seção de planejamento que inclui a metodologia a ser seguida, os produtos finais esperados, os recursos necessários, as etapas do trabalho e um cronograma.

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 à 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. Os trabalhos sobre STR encontrados na bibliografia podem ser classificados, com razoável consistência, com respeito a perspectiva adotada. São quatro as questões ligadas à tempo real mais discutidas na literatura:

- Paradigmas de Programação e Linguagens de Programação: Nesta categoria então os trabalhos envolvendo paradigmas e linguagens de programação para a construção de STR. Neste sentido, são importantes as abstrações usadas no momento de programar o sistema. Os paradigmas de programação mais empregados são o baseado em processos e a orientação à objetos. Existem alguns trabalhos visando adaptar também a programação lógica para o contexto de tempo real. Dentro do paradigma de programação escolhido, a temporalidade do sistema pode ser associada com algorítmos (existem prazos para executar os algorítmos), com dados (existem prazos de validade para os dados) ou ainda com a resposta do sistema (existem prazos para responder à eventos externos).

- Escalonamento: Nesta categoria estão os trabalhos preocupados em descrever como a execução do sistema acontece, na ótica do 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 algorítmos 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.

- Organização do Software: Nesta categoria estão os trabalhos que descrevem como o software de um STR pode ser organizado. As abstrações mais importantes neste ponto de vista são os componentes de software e suas interfaces. Os conceitos encontrados com maior frequência são: camadas de software, serviços, interfaces, núcleos, sistemas operacionais, etc. Normalmente, as soluções propostas dentro deste ponto de vista recebem o nome genérico de "suportes para sistemas tipo tempo real".

- Arquitetura do Hardware: Nesta categoria estão os trabalhos que analisam e/ou propõe arquiteturas de hardware mais apropriadas para aplicações com restrições de tempo real. Por exemplo, nas aplicações convencionais, o mais importante é o comportamento temporal do hardware no caso médio. Em sistemas tipo tempo real, o comportamento no pior caso é tão ou mais importante do que o caso médio. Outro aspecto é o suporte do hardware para a sincronização de relógios em ambientes distribuídos.

Um ambiente completo para a construção de sistemas tipo tempo real deve apresentar soluções, coerentes entre si, para todas estas questões. Embora as quatro perspectivas estejam obviamente relacionados entre si, é possível concentrar o trabalho em apenas uma, ignorando as demais. Nesta proposta de tese, a perspectiva predominante será aquela relacionada com o escalonamento.

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 à sistemas em software, pode ser encontrada em [MOT 93]. Referências adicionais à trabalhos publicados na área de tempo real podem ser encontradas em [AUD 90c], [RAM 94] e [SHI 94].

No restante do capítulo serão apresentados os conceitos básicos de sistemas tempo real. Também será feita uma revisão das principais soluções encontradas na bibliografia para o problema do escalonamento tempo real. Ao longo do capítulo serão discutidas as abstrações encontradas ao nível do escalonamento. Neste nível de abstração, serão ignorados os aspectos relacionados com as linguagens de programação. Especificamente, não serão discutidas as construções sintáticas e semânticas necessárias para a construção de Sistemas Tempo Real.

2.2 Conceitos Básicos

Esta seção define alguns conceitos básicos ligados à sistemas de tempo real. 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 algorítmo 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 habilitação, 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 habilitações sucessivas.

O momento de ativação de uma tarefa aperiódica (esporádica ou não) é chamado de momento de chegada ("arrival time") ou de liberação ("release time"). Neste momento, o escalonador toma conhecimento da necessidade de executar a tarefa. Neste texto, momento de chegada e momento de liberação serão tratados como sinônimos. Ambos são bastante empregados na literatura.

O momento de pronto ("ready time") de 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 pronto 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 do momento de pronto.

A maioria dos modelos que tratam com tarefas periódicas consideram o início de cada período como o momento de pronto daquela ativação em particular da tarefa, ou seja, o deslocamento ("offset") é sempre zero com relação ao início do período. Alguns modelos mais elaborados permitem que exista um deslocamento ("offset") maior que zero entre o início do período e o momento de pronto da tarefa.

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

- Período ***P*** (tarefas periódicas);
- Intervalo de tempo entre habilitações ***I*** (tarefas esporádicas);
- Tempo 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 ***(C1,C2)***, onde ***C1*** é o tempo de

computação necessário dentro do prazo de execução e ***C2*** é o tempo das computações da tarefa que podem ser realizadas fora (depois) do prazo de execução.

Algumas aplicações definem ainda um jitter[1] máximo ("maximum jitter") para cada tarefa periódica. Ainda que a 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. Alguns algorítmos para controle de processos são sensíveis ao nível de jitter que a tarefa experimenta.

Um conceito semelhante ao jitter máximo é o jitter na liberação ("release jitter"). Isto acontece sempre que existem tarefas esporádicas mas o sistema somente amostra eventos externos em momentos pré-determinados (por exemplo, a cada tick do relógio). 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 um tick do relógio, ele somente será reconhecido, para efeitos de escalonamento, no próximo tick. 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 na liberação de ***J*** unidades de tempo. A figura 2.1 ilustra este conceito.

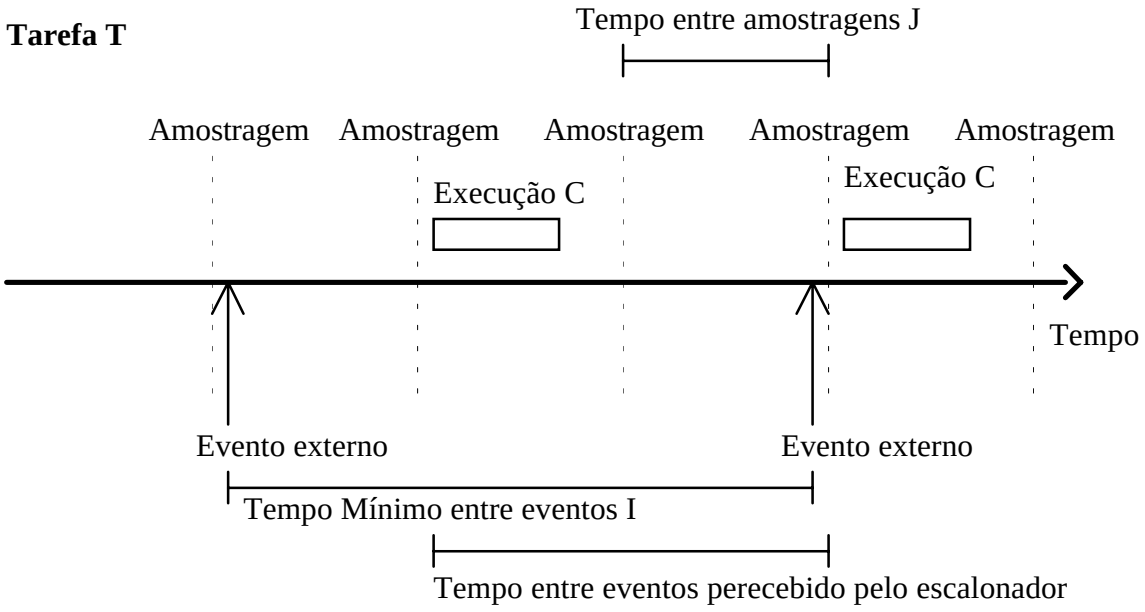

Figura 2.1 - Linha de tempo ilustrando o jitter na liberação.

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. Alguns algorítmos de escalonamento utilizam

[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.

o conceito de instante crítico para determinar se um conjunto de tarefas é ou não escalonável. Se existir, 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 ***Ti*** precede a tarefa ***Tj***, então a tarefa ***Tj*** somente pode iniciar sua execução após o término da tarefa ***Ti***. Em geral, o início da atividade está associado à 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 à uma resposta do sistema para o ambiente. A figura 2.2 ilustra uma aplicação com 2 atividades e 7 tarefas. Na figura é possível observar que a tarefa ***T4*** é precedida pelas tarefas ***T2*** e ***T3***.

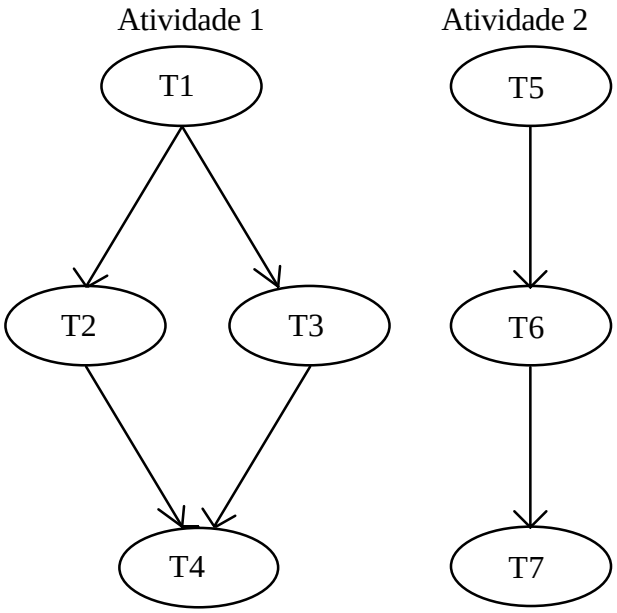

Figura 2.2 - Duas atividades, com 7 tarefas.

Uma questão chave é como expressar o deadline para execução de uma tarefa. A situação mais simples é ilustrada na figura 2.3. 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 habilita 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.

Em sistemas maiores, especialmente os distribuídos, definir os deadlines das tarefas é mais complicado. A figura 2.4 ilustra uma situação que pode ocorrer, por exemplo, em uma planta industrial envolvendo controle hierarquizado. Neste caso, um estímulo do ambiente habilita 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 da margem à diferentes maneiras de expressar o "prazo satisfatório" de uma tarefa.

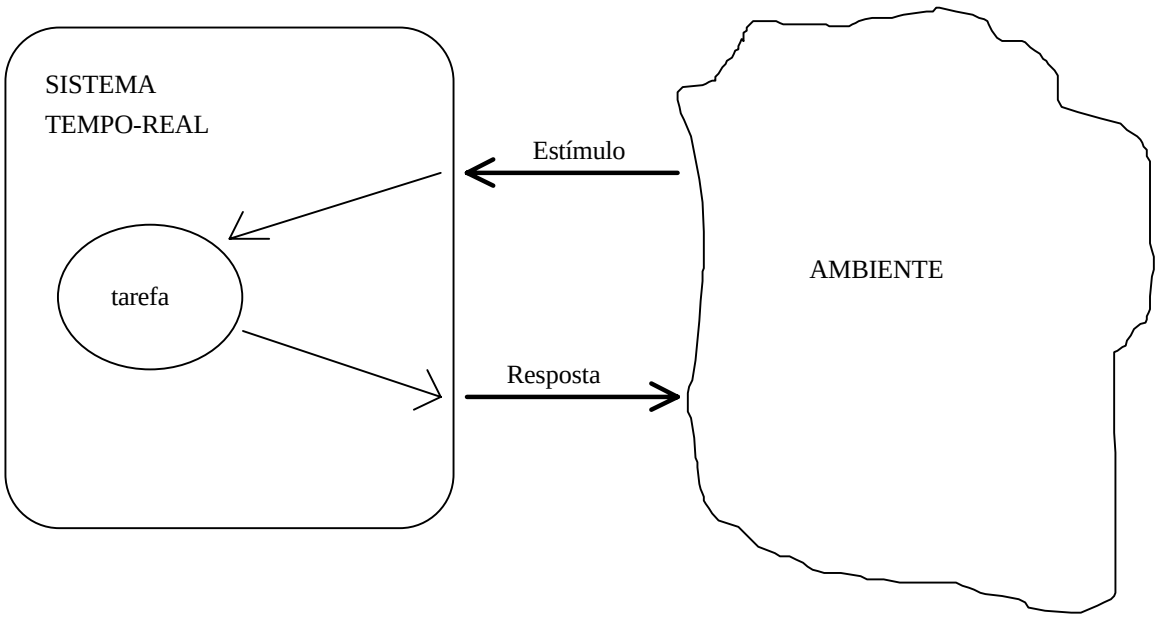

Figura 2.3 - Sistema simples, uma única tarefa responde ao ambiente.

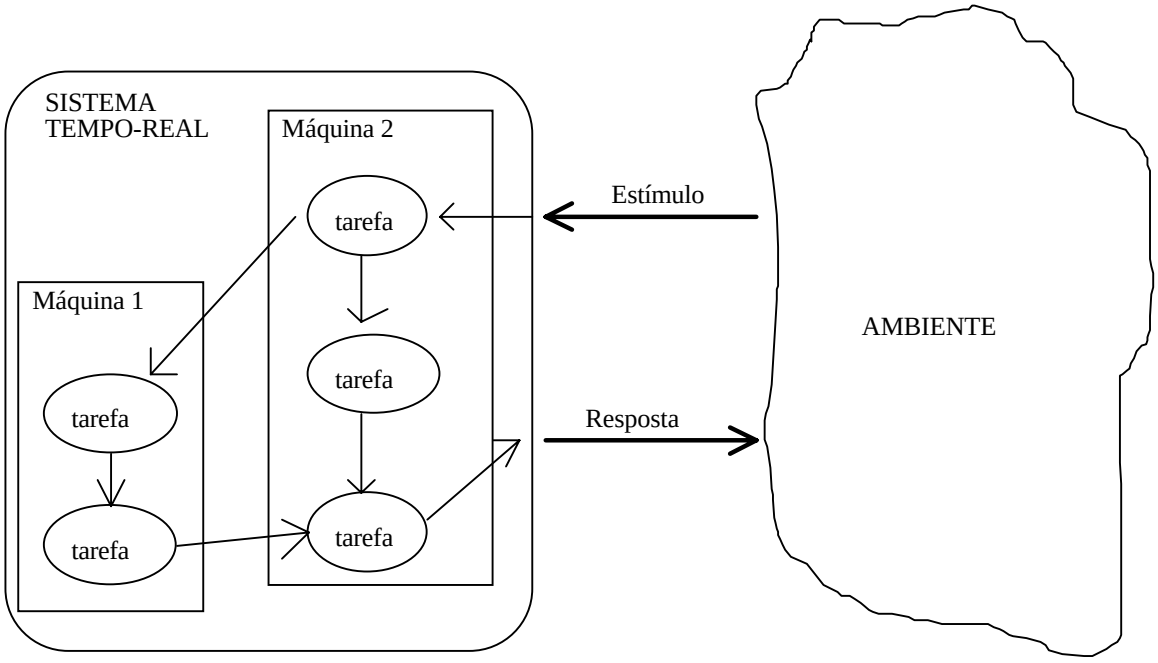

Figura 2.4 - Sistema complexo, um grafo de tarefas responde ao ambiente.

2.2.2 Propriedades dos Modelos de Tarefas

O modelo de tarefas de uma aplicação tempo real descreve como acontece a execução das tarefas da aplicação. É considerado apenas o comportamento do sistema com respeito a utilização dos recursos computacionais no tempo. Na verdade, é possível dividir o modelo em duas partes. Uma delas representa os "usuários de recursos" (as tarefas e suas propriedades) e a outra representa os "recursos do sistema" (processadores, meio de comunicação, etc). Alguns autores fazem distinção entre modelo de tarefas, que descreve as tarefas, e modelo de recursos, que descreve os recursos. Neste texto, o conceito modelo de tarefas será utilizado com um sentido amplo, englobando os dois aspectos.

Nesta seção, será feita uma revisão das principais propriedades presentes nos modelos de tarefas encontrados na bibliografia. É preciso lembrar que o modelo de tarefas é influenciado pela abordagem escolhida para a questão da previsibilidade. Algumas propriedades fazem sentido em uma abordagem mas não em outra. A diversidade destas propriedades vem no sentido de atender a grande a variedade de tipos de aplicações tempo real existentes.

Criação dinâmica de tarefas

Alguns modelos de tarefas admitem a criação dinâmica de tarefas durante a execução. Outros modelos exigem que todas as tarefas já estejam criadas em tempo de projeto (criação estática de tarefas).

Periodicidade das tarefas

Como colocado antes, tarefas podem ser periódicas, aperiódicas esporádicas ou aperiódicas não-esporádicas. Com respeito a este aspecto, alguns modelos aceitam somente tarefas com ativações periódicas. Modelos um pouco mais flexíveis aceitam tarefas com ativações periódicas ou aperiódicas esporádicas. Existem ainda modelos capazes de trabalhar com tarefas aperiódicas em geral.

Deslocamento de fase em tarefas periódicas ("offset")

A maioria dos modelos que tratam com tarefas periódicas consideram o início de cada período como o momento da habilitação da tarefa, ou seja, deslocamento é sempre zero com relação ao início do período. Alguns modelos mais elaborados permitem que exista um deslocamento ("offset") maior que zero entre o início do período e a habilitação da tarefa.

Escopo dos deadlines

Com respeito ao escopo dos deadlines, estes podem estar associados à tarefas individualmente ou estar associados à atividades (grupos de tarefas).

Precedência entre tarefas

Alguns modelos de tarefas suportam uma relação de precedência entre tarefas. Por exemplo, a tarefa ***P*** precede a tarefa ***Q*** se e somente se a tarefa ***Q*** somente pode executar após a tarefa ***P*** terminar sua execução. Se ***P*** e ***Q*** são periódicas, então elas devem possuir o mesmo período e cada ativação de ***P*** precede uma respectiva ativação de ***Q***. Modelos de tarefas mais simples não admitem a relação de precedência entre tarefas.

Recursos além dos processadores

Alguns modelos de tarefas conseguem lidar com a exclusão mútua entre tarefas da aplicação, resultado da disputa por recursos além dos processadores. Modelos mais simples consideram todas as tarefas como independentes, ou seja, o único recurso computacional considerado são os processadores.

Distribuição do sistema

Os modelos de tarefas mais simples admitem apenas um único computador. Alguns modelos de tarefas são criados especialmente para arquiteturas distribuídas, ou seja, um conjunto de computadores interligados através de uma rede de comunicação (acoplamento fraco). Entre estes últimos, alguns trabalham com um conjunto de computadores homogêneos, enquanto outros suportam um conjunto de computadores heterogêneos.

Paralelismo real

Os modelos de tarefas mais simples supõe que cada computador contém um único processador. Alguns modelos de tarefas são criados especialmente para arquiteturas paralelas. Os modelos de tarefas mais gerais admitem que cada computador contém diversos processadores, cada um podendo ser utilizado como for mais conveniente (possibilidade de paralelismo real entre as tarefas da aplicação).

Restrições à alocação das tarefas

Com respeito às restrições que podem ser colocadas à alocação de tarefas em processadores, os modelos mais simples consideram que as tarefas são independentes no momento da alocação. Modelos mais elaborados permitem restrições do tipo:
- A aplicação pode exigir que duas tarefas nunca sejam executadas no mesmo processador ou computador (alocação excludente);
- A aplicação pode exigir que duas tarefas sempre sejam executadas no mesmo processador ou computador (alocação simultânea);

Meio de comunicação

Em arquiteturas distribuídas o meio de comunicação entre os computadores é um aspecto relevante. Abaixo estão as situações mais usuais:

- Meio de comunicação com tempo de acesso e transmissão determinista, independente do fluxo de dados;
- Meio de comunicação com tempo de acesso e transmissão probabilista, independente do fluxo de dados;
- Meio de comunicação com tempo de acesso e transmissão determinista, dependente do fluxo de dados;
- Meio de comunicação com tempo de acesso e transmissão probabilista, dependente do fluxo de dados;
- Meio de comunicação deve ser escalonado.

2.2.3 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. 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. Neste caso a escala descreve execuções parcias da tarefa que estão 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 qual slot 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.

Neste trabalho, tanto o conceito de previsibilidade determinista como o de previsibilidade probabilista serão considerados válidos, desde que adequados para a aplicação em questão. Também será considerada válida a exigência de uma previsibilidade determinista para quaisquer tarefas que sejam críticas para a missão do sistema.

Teste de escalonabilidade e Cálculo da escala de execução

O escalonamento de um conjunto de tarefas é, muitas vezes, dividido em duas etapas. Inicialmente, um teste de escalonabilidade determina se é possível atender os deadlines das tarefas. A segunda etapa corresponde à 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.

Os algorítmos que realizam testes de escalonabilidade presentes na literatura (como ilustra a figura 2.5) 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.

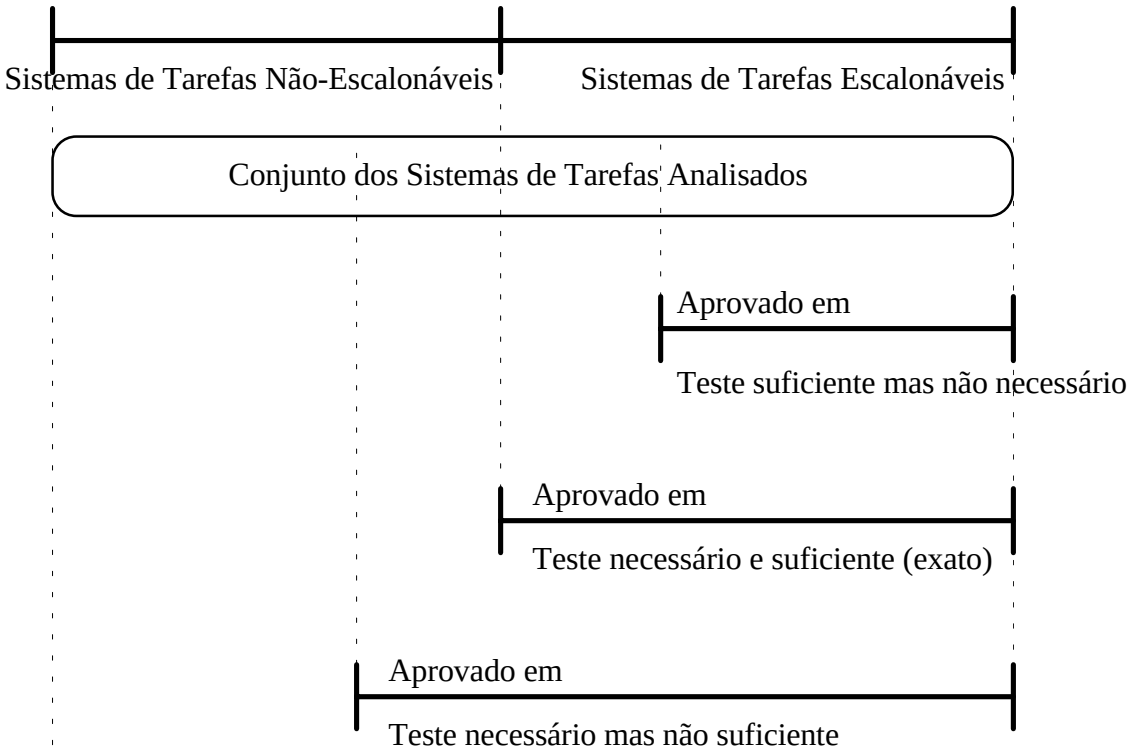

Figura 2.5 - Classificação dos teste de escalonabilidade.

Utilização do processador

Algumas soluções de escalonamento estão baseadas no conceito de utilização ("utilization") do processador. A utilização do processador ***Ui***, referente a tarefa ***Ti***, 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 ***Ui***, referente a tarefa ***Ti***, é definida como:

Ui* = *Ci* / *Pi, para tarefas periódicas;
Ui* = *Ci* / *Ii, para tarefas esporádicas.

Onde:

- ***Ci*** é o tempo máximo de computação da tarefa ***Ti***;
- ***Pi*** é o período da tarefa ***Ti*** (tarefas periódicas);
- ***Ii*** é o intervalo mínimo entre ativações da tarefa ***Ti*** (tarefas esporádicas).

Embora este conceito não seja normalmente aplicado às tarefas aperiódicas, a utilização do processador referente à uma tarefa aperiódica é, por definição, infinita.

A utilização total do processador ***U***, devido ao conjunto de ***N*** tarefas, é definida como:

$$U = \sum_{i=1,2,...,N} (Ui)$$

Carga estática e Carga dinâmica

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*

Nas próximas seções serão apresentadas 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.

Escalonamento em ambiente distribuído

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. Em sistemas com requisitos de tempo real não é normalmente possível a migração de tarefas em tempo de execução. Logo, cada tarefa é alocada permanentemente à um processador. 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.

2.3 Classificação das Abordagens para o Escalonamento Tempo Real

Na literatura são identificadas diferentes propostas de como sistemas tempo real podem ser construídos. Estas propostas muitas vezes diferem consideravelmente. Entretanto, alguns aspectos são relevantes para qualquer tentativa de classificação: Abordagem com respeito a previsibilidade, utilização de recursos e algorítmos de escalonamento empregados.

A construção de um sistema tempo real implica em fazer escolhas com respeito às três classes de aspectos. Primeiramente, é preciso decidir que tipo de previsibilidade é necessária para o sistema a ser implementado. Entre as abordagens capazes de satisfazer os requisitos da especificação, pode ser escolhida aquela que melhor utiliza os recursos do sistema. As propriedades do modelo de tarefas adotado devem suportar a abordagem escolhida. Não adianta, por exemplo, associar às tarefas um tempo médio de execução se a abordagem escolhida pretende garantir o cumprimento dos prazos no pior caso. Finalmente, devem ser escolhidos algorítmos de escalonamento compatíveis com o modelo de tarefas adotado e os objetivos da abordagem escolhida.

A figura 2.6 resume os conceitos que serão discutidos nesta seção. Especificamente, mostra as 5 abordagens básicas para a construção de Sistemas Tempo Real. Estas abordagens serão descritas abaixo.

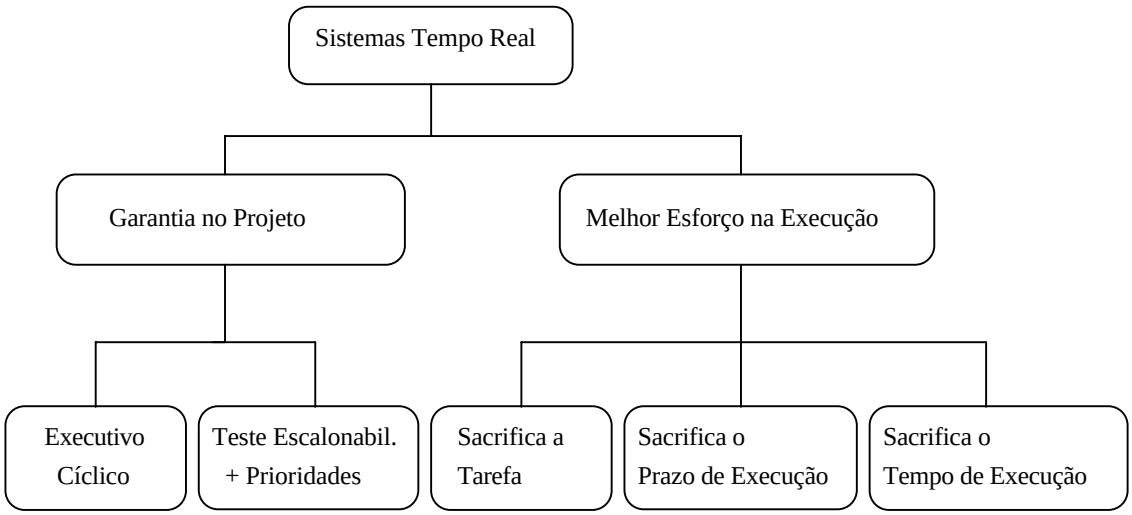

Figura 2.6 - Possíveis abordagens para sistemas tempo real.

Como definido antes, escalonamento estático é aquele capaz de oferecer uma previsibilidade determinista. Ele é empregado em sistemas onde é necessário garantir, em tempo de projeto, que todas as tarefas serão executadas dentro dos 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"). As condições normalmente aceitas para tal garantia envolvem:

- Que a carga seja estática, ou seja, limitada e conhecida em tempo de projeto;
- Reservar recursos para a execução de todas as tarefas no pior caso.

Uma forma utilizada para garantir deadlines em tempo de projeto é calcular, ainda em tempo de projeto, o que cada processador vai fazer a cada instante de tempo durante a execução do sistema. Todo o trabalho de escalonamento é realizado no projeto. O resultado é uma grade ("time grid") que determina "qual tarefa roda quando em qual processador". Esta solução é chamada de executivo cíclico estático, pois um pequeno programa de controle (executivo) repete esta grade ciclicamente, durante toda a vida do sistema. A complexidade dos algorítmos utilizados para calcular a grade depende basicamente das propriedades do modelo de tarefas.

Nos últimos anos, resultados teóricos importantes foram obtidos a partir de algorítmos de escalonamento que trabalham com prioridades, especialmente prioridades fixas. Nestes algorítmos, cada tarefa recebe uma prioridade fixa em tempo de projeto. Um teste de escalonabilidade, também em tempo de projeto, determina se existe a garantia de que todas as tarefas serão executadas dentro dos deadlines. Em tempo de execução, um escalonador preemptivo executa as tarefas habilitadas conforme as suas prioridades. Também existem propostas dentro desta classe que trabalham com prioridades variáveis, mas os resultados teóricos são menos significativos. As propostas que empregam prioridades se diferenciam em função das propriedades do modelo de tarefas adotado e dos testes de escalonabilidade envolvidos.

A grande vantagem do escalonamento estático é oferecer uma previsibilidade determinista para o cumprimento dos prazos das tarefas. Para isto, é necessário uma reserva de recursos para o pior caso. Isto pode representar uma enorme subutilização de recursos. Por exemplo, o tempo médio de execução de determinados algorítmos é consideravelmente menor do que o tempo de execução no pior caso. Ao mesmo tempo, um sistema que emprega escalonamento estático não aproveita o fato de muitas aplicações admitirem que prazos de execução sejam eventualmente perdidos. Um outro problema com esta abordagem é a exigência de uma carga limitada e estática. A adaptabilidade de um dado sistema será maior se ele for capaz de disparar novas tarefas em reação aos eventos observados no ambiente. Em resumo, no escalonamento estático existe o sacrificio de recursos e flexibilidade com o objetivo de obter previsibilidade.

Nas abordagens que utilizam escalonamento dinâmico, não existe garantia, em tempo de projeto, de que os deadlines serão cumpridos. Estes sistemas também são chamados de "melhor esforço" ("best effort"), pois fazem o possível para cumprir os prazos de execução, sem oferecer garantia. Quando muito, é possível obter uma previsibilidade probabilista para o comportamento temporal do sistema, a partir de uma estimativa também probabilista 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, algo que inexiste no escalonamento estático. 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 do sistema, mas uma situação que ocorre naturalmente em sistemas que empregam uma abordagem tipo melhor esforço. Logo, é necessário um mecanismo para tratar a sobrecarga. Existem três procedimentos básicos:

- Descarta completamente algumas tarefas;
- Executa todas as tarefas, mas sacrifica o prazo de execução de algumas;
- Executa todas as tarefas, mas sacrifica o tempo de execução de algumas.

No tratamento da sobrecarga é preciso definir quais tarefas serão sacrificadas. O mecanismo é executado ou no sentido de maximizar uma função benefício do sistema, ou minimizar uma função erro. Por exemplo, se o mecanismo utilizado é o descarte de tarefas, poderão ser descartadas as tarefas menos importantes para a aplicação. Obviamente, a função benefício/erro de um sistema é totalmente dependente da aplicação em questão. Ela faz parte do modelo de tarefas utilizado. Em geral, funções benefício/erro mais elaboradas conseguem representar melhor a realidade da aplicação, mas aumentam o custo do escalonamento. Em [JEN 93], Jensen leva este conceito ao extremo propondo que cada tarefa possua uma função benefício/tempo tão elaborada quanto se queira, determinando o escalonamento das tarefas a partir destas funções.

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 à 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 algorítmos 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.

A classificação das possíveis abordagens, apresentada nesta seção, é muito semelhante à apresentada em [RAM 94]. A única diferença está na inclusão da abordagem "Sacrifica o Tempo de Execução", não presente naquele artigo.

2.4 Exemplificação da Taxonomia Apresentada

Na seção 2.3 foi apresentada uma classificação das principais abordagens para a construção de STR. Nesta seção aparecem exemplos de algorítmos de escalonamento e seus modelos de tarefas. Não existe a pretenção de uma listagem exaustiva das propostas existentes, pois isto exigiria centenas de páginas. O objetivo desta seção é citar propostas que ilustram as diferentes linhas identificadas na classificação apresentada.

As propostas estão agrupadas conforme a classe na qual melhor se encaixam. Dentro de cada classe apresentada, muitas propostas extendem propostas anteriores, normalmente flexibilizando o modelo de tarefas. Em particular, soluções para ambientes distribuídos são geralmente extensões de soluções anteriores para ambientes centralizados. Sempre que possível, a mesma notação e terminologia foi utilizada na descrição de todas as propostas. Embora às vezes a notação utilizada aqui seja diferente daquela utilizada pelo autor da proposta. Uma notação única torna mais fácil para o leitor comparar as propostas.

2.4.1 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 algorítmos 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 e deslocamento. Tarefas esporádicas são transformadas em periódicas[2]. 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 algorítmo empregado. Tabelas resumem os resultados.

Nos próximos parágrafos, são descritas algumas propostas encontradas na literatura que exemplificam esta classe de abordagem. Outros exemplos poderão ser encontrados em [XU 93a].

Sistema Mars

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. Neste texto, será descrita apenas a abordagem para o escalonamento tempo real.

Como dito antes, o sistema deve ser projetado em cima de análises de pior caso. O projeto de uma aplicação é orientado à atividades ("transactions"). A comunicação entre tarefas é feita através de mensagens de estado ("state messages"), com semântica semelhante à uma variável global. Uma nova versão de uma mensagem de estado sobrepõe-se à versão anterior. Uma mensagem de estado não é consumida pela leitura, mas possui um prazo de validade explícito.

Uma aplicação no sistema Mars é formada a partir de um conjunto de ***M*** atividades ("transactions") periódicas. Cada atividade ***Ai***, ***i*** entre ***1*** e ***M***, possui associado um período ***Pi*** ("duty cicle" ou "minimal interarrival time") e um deadline ***Di*** ("maximal response time"). Cada atividade é representada por um grafo dirigido acíclico, correspondendo a um

[2]No pior caso, uma tarefa esporádica comporta-se como uma tarefa periódica com período igual ao tempo mínimo entre ativações.

conjunto de tarefas que se comunicam. No sistema Mars, os nodos desde grafo representam tarefas e os arcos representam mensagens de estado. Cada tarefa ***Tj*** possui um tempo máximo de execução ***Cj*** conhecido ("maximal execution time") e admite preempção a qualquer momento (pode ser interrompida e continuada mais tarde). Todas as tarefas pertencentes a uma mesma atividade ***Ai*** compartilham o seu período ***Pi***. A deadline ***Di*** da atividade está associada com a conclusão de todas as tarefas pertencentes a atividade.

O hardware é composto por componentes interligados através de um barramento, onde cada componente é formado por um computador completo e suas respectivas tarefas. São usados algorítmos que provêm uma base de tempo global, cujo sincronismo entre componentes possui uma precisão conhecida. A figura 2.7 ilustra uma atividade cujas tarefas foram distribuídas por três componentes.

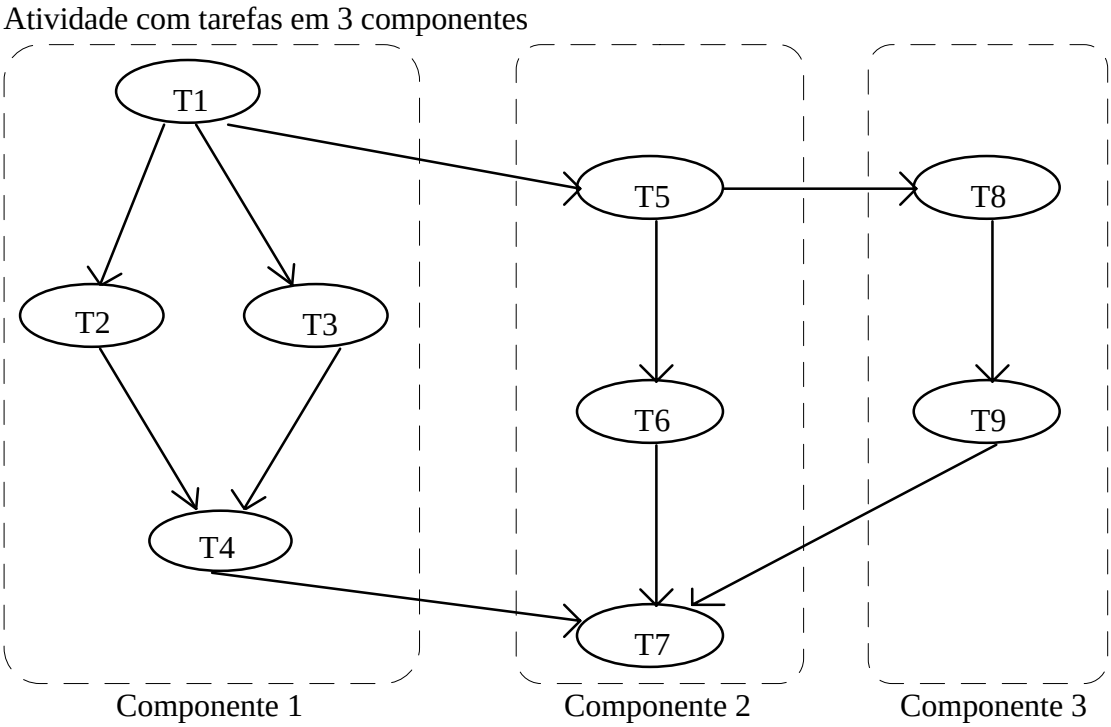

Figura 2.7 - Um exemplo de atividade no Sistema Mars.

Existe um controle direto da aplicação sobre o tráfego no meio de comunicação, sem redundância dinâmica. A comunicação é feita através de datagramas sem confirmação. O acesso ao meio de comunicação é feito com TDMA ("Time Division Multiple Acess"). Este método foi escolhido por ser determinista e livre de colisão.

O trabalho de escalonamento é dividido em duas etapas. Primeiramente, as tarefas são alocadas aos componentes (sistema distribuído). Uma mesma atividade pode ter suas tarefas distribuídas por mais de um componente. A partir de uma dada alocação, é determinado o escalonamento local dos processadores (um em cada componente) e dos slots TDMA do barramento. Os slots do meio de comunicação são recursos cuja gerência é integrada com a gerência do recurso processador.

Em Mars, a alocação das tarefas aos processadores pode ser feita manualmente ou através de pesquisa heurística sub-ótima. Pode ser também utilizada uma forma mista, onde parte da alocação é feita manualmente (para atender à critérios específicos do projeto) e o restante é feito de maneira automática.

O escalonamento dos processadores e dos slots TDMA é feito através de uma pesquisa heurística sub-ótima ([KOR 85]). Uma função heurística calcula a urgência de uma tarefa em função de estimativas do tempo necessário para completar a atividade ("transaction"). Esta função heurística determina qual a parte da árvore total de soluções a ser inspecionada. Esta pesquisa leva em conta as relações de precedência entre tarefas. Não existem situações de exclusão mútua entre tarefas, pois o único mecanismo de comunicação entre tarefas disponível são as mensagens de estado. Uma mensagem de estado está sempre disponível para leitura e jamais é consumida, não gerando situações de bloqueio.

É construída uma grade de execução para cada processador e mais uma grade para os slots do barramento. Todas as grades geradas (escalonamentos) possuem a duração de um MMC (mínimo múltiplo comum) dos períodos das atividades. Tanto a ocupação dos processadores como a do suporte de comunicação é feita por um escalonamento dirigido pelo tempo ("time-trigger"). A base para este escalonamento dirigido pelo tempo é o tempo global conseguido pela sincronização dos relógios físicos. Este modelo é identificado na literatura como síncrono ([LAM 84]), tanto a nível de processamento como de comunicação.

Em [FOH 92] é feita uma descrição do algorítmo usado, assim como uma análise qualitativa a respeito da influência de propriedades da aplicação sobre o desempenho do algorítmo. O objetivo em [FOH 92] é determinar quais são as propriedades da aplicação que tornam uma instância específica do problema mais ou menos difícil, em termos de tempo de execução do algorítmo.

Outras propostas

Em [AGN 91] são apresentados três diferentes algorítmos para o cálculo de grades de execução. Esta solução é chamada no artigo de escalonamento cíclico global. O modelo de tarefas empregado pressupõe um sistema distribuído composto por diversos nodos monoprocessadores. O meio de comunicação suporta tanto mensagens simples como difusões. Os algorítmos apresentados geram uma grade de execução para cada processador e mais uma para o meio de comunicação. Todas as grades são do mesmo tamanho, equivalente ao MMC (mínimo múltiplo comum) de todos os períodos da aplicação. Elas devem ser executadas simultaneamente. É necessária uma sincronização de relógio entre todos os nodos da rede.

Um outro modelo é apresentado em [VER 91], onde é utilizado um modelo de tarefas que inclui tarefas com deadlines e relações de precedência. É suposta a existência de outros recursos, além dos processadores, que exigem acesso exclusivo. O sistema é distribuído, com máquinas homogêneas e um atraso constante associado com o envio de messagens entre computadores. Uma heurística sub-ótima é empregada no cálculo das grades de execução para um período equivalente ao mínimo múltiplo comum dos períodos das tarefas.

Em [XU 93b] é descrito um modelo que considera um conjunto de atividades ("processes") formadas por tarefas ("segments") que podem possuir relações de precedência e exclusão entre si. Deadlines são associados com as atividades e as tarefas não podem ser "preemptadas". O artigo apresenta um algorítmo que busca uma grade de execução para um multiprocessador que atenda os requisitos do problema. É feita uma pesquisa exaustiva no espaço de soluções, com heurísticas que dirigem o caminhamento.

Em [FOH 94] a abordagem baseada em executivo cíclico é extendida no sentido de oferecer alguma adaptabilidade em tempo de execução. Basicamente, Fohler cria mecanismos dentro do executivo cíclico para viabilizar trocas de modo de operação ("mode change") e aproveitamento da capacidade ociosa ("spare time") para execução de tarefas aperiódicas não garantidas. Desta forma, o executivo cíclico é capaz de oferecer a mesma flexibilidade que as propostas baseadas em prioridades.

Além das propostas descritas nesta seção, inúmeros outros artigos na bibliografia apresentam soluções semelhantes para o problema do escalonamento, seguindo a abordagem baseada em executivo cíclico. Alguns exemplos podem ser encontradas em [HOU 92], [LAW 92], [YUA 94] e [BLA 94].

2.4.2 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 as 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 algorítmo Taxa Monotônica (RM - "rate monotonic") trabalha com prioridades fixas e associa prioridades mais altas para as tarefas com menor período. O algorítmo 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 algorítmos 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. Surgiram

outros algorítmos além do EDF e do RM. Esta abordagem aparece hoje como uma alternativa ao executivo cíclico, descrito na seção anterior. Nos próximos parágrafos será resumido o estado atual das propostas que seguem esta abordagem.

2.4.2.1 Taxa Monotônica

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].

Modelo Básico

No modelo de tarefas básico associado com RM, existem ***N*** tarefas periódicas e independentes, compartilhando um único processador. Cada tarefa ***Ti***, com ***i*** entre ***1*** e ***N***, é caracterizada por um período ***Pi***, um tempo máximo de computação ***Ci*** e um deadline ***Di*** igual ao período. O algorítmo RM trabalha com prioridades fixas e associa prioridades superiores para as tarefas com menor período.

Em [LIU 73] é descrito um teste de escalonabilidade suficiente mas não necessário, baseado no conceito de utilização ("utilization") do processador. A utilização do processador ***Ui***, referente a tarefa ***Ti***, é definida como:

$$Ti = Ci / Pi.$$

A utilização total do processador ***U***, devido ao conjunto de ***N*** tarefas, é definida como:

$$U = \sum_{i=1,2,\ldots,N} (Ui)$$

O teste de escalonabilidade descrito em [LIU 73] limita a utilização do processador em 0.69, para valores grandes de ***N***. Este é um limite pessimista. Em [LEH 89] é mostrado que, na prática, a utilização que pode ser garantida chega em média a 88%. Um teste de escalonabilidade exato é descrito em [LEH 89] e [SHA 89].

Extensões ao Modelo Básico

Uma das primeiras extensões feitas sobre o modelo original foi permitir a execução de tarefas esporádicas. Isto é feito através de servidores periódicos como, por exemplo, o servidor esporádico ("sporadic server") descrito em [SPR 89].

Quando o modelo de tarefas é ampliado para permitir exclusão mútua (seções críticas) entre tarefas, surge o fenômeno conhecido como inversão de prioridade ("priority inversion"). Considere uma aplicação com 3 tarefas ***T1***, ***T2*** e ***T3***, onde ***T1*** possui a prioridade mais alta e ***T3*** a mais baixa. Suponha que ***T1*** e ***T3*** compartilham um recurso que exige exclusão mútua no acesso. Este recurso pode ser um dispositivo ou uma estrutura de dados global. Considere a seguinte sequência de eventos, partindo da situação na qual o processador está livre:

- ***T3*** é liberada, inicia a executar e acessa o recurso compartilhado;
- ***T1*** é liberada, obtém o processador e inicia a executar;
- ***T1*** tenta acessar o recurso que está ocupado, bloqueia, T3 volta a executar;
- ***T2*** é liberada, obtém o processador e inicia a executar.

Na situação descrita acima, a tarefa ***T1*** é obrigada a esperar a execução da tarefa ***T2***, embora ***T1*** possua maior prioridade e não compartilhe nenhum recurso com ***T2***. Nesta situação, as prioridades de ***T1*** e ***T2*** parecem invertidas.

Existem na bibliografia algumas propostas de solução para este problema ([DAV 91]). A solução normalmente empregada em RM é um algorítmo chamado protocolo da prioridade máxima (PCP - "priority ceiling protocol"), descrito em [SHA 90]. Este algorítmo impede a ocorrência de deadlocks. Ele também limita o tempo de bloqueio sofrido por uma tarefa, em função de recursos ocupados por tarefas de menor prioridade. Este tempo de bloqueio é no máximo igual ao tempo de execução da mais longa seção crítica de uma tarefa com prioridade inferior. Este tempo de bloqueio pode ser facilmente incluído no teste de escalonabilidade.

Alguns autores destacam que RM permite um certo controle sobre situações de sobrecarga. Nestas situações, algumas tarefas não teriam os seus deadlines cumpridos em função de alguma falta de projeto. Por exemplo, o tempo máximo de execução de uma determinada tarefa ter sido subestimado para efeito do teste de escalonabilidade. É perfeitamente possível no RM que uma tarefa de maior importância receba uma prioridade inferior, por ter um período maior que tarefas menos importantes. Neste caso, a tarefa mais importante é a que vai perder o seu deadline. É possível aumentar a prioridade da tarefa mais importante sem comprometer o teste de escalonabilidade. Basta dividir a sua computação em ***K*** partes consecutivas e dividir o seu período por ***K***. A cada novo período, uma das partes é executada. A cada ***K*** novos períodos, temos um período original completo e a tarefa original executada completamente. A redução do seu período fará com que as prioridades de suas divisões aumente, seguindo a política do RM. Esta técnica é conhecida como transformação de período ("period transformation").

Ainda em [SHA 94] é mostrado como o modelo de tarefas do RM é extendido para o ambiente distribuído. Os autores discutem o que é necessário para um protocolo de comunicação garantir prazos para o envio de mensagens através de uma rede de comunicação. A partir da existência de tais protocolos, o artigo ilustra um método ad hoc para a alocação de tarefas em processadores e o particionamento dos deadlines. Feito isto, RM é aplicado a cada processador individualmente. O particionamento dos deadlines é necessário quando uma atividade inclui tarefas em diversos processadores e, em consequëncia, comunicação entre eles. Para que RM possa ser aplicado em cada processador de maneira independente, o prazo da atividade como um todo deve ser particionado em prazos menores para cada tarefa ("partition of end-to-end deadlines").

Diversos outros artigos tratam da proposta RM. Por exemplo, em [SHA 89] é analisado o problema da mudança de modo operacional ("mode change") mantendo a garantia da escalonabilidade. Em [SHI 93] o algorítmo RM é modificado para a situação na qual o deadline das tarefas corresponde à um valor múltiplo do período ("deferred deadline"). Neste caso, existe a possibilidade de diversas ativações simultâneas da mesma

tarefa. Em [KAT 93] é analisado o impacto sobre algorítmos que trabalham com prioridade fixa dos custos associados com a execução do escalonador.

2.4.2.2 Próximo deadline

Como dito anteriormente, no algorítmo Próximo Deadline as tarefas possuem prioridades variáveis. Recebe maior prioridade a tarefa cujo deadline está mais próximo do momento atual.

Modelo Básico

A proposta original do algorítmo EDF define um modelo de tarefas com ***N*** tarefas periódicas e independentes. Cada tarefa ***Ti*** é caracterizada por um período ***Pi***, um tempo máximo de computação ***Ci*** e um deadline ***Di*** igual ao período. Da mesma forma que no RM, cada tarefa ***Ti*** possui associada uma utilização do processador ***Ui.*** A utilização total do processador ***U***, devido ao conjunto de ***N*** tarefas, é o somatório das respectivas ***Ui***.

Neste modelo de tarefas, um conjunto de tarefas é escalonável se a utilização total do processador ***U*** for inferior à 100%. A utilização do processador possível com EDF é excelente, mas sua aplicação é limitada pelo modelo de tarefas muito simples.

Extensões ao Modelo Básico

Algumas extensões ao modelo básico do EDF podem ser encontradas na bibliografia. Em [CHE 90b], os autores extendem o protocolo da prioridade máxima (PCP - "priority ceiling protocol") para um esquema de prioridades variáveis, compatível com o EDF. Desta forma, o modelo de tarefas do EDF passa a incluir recursos além do processador. É mostrado que o novo protocolo preserva todas as propriedades desejáveis presentes no PCP, ou seja, um tempo máximo de bloqueio limitado e ausência de deadlock. O artigo ainda apresenta um teste de escalonabilidade que leva em consideração o protocolo apresentado, permitindo uma análise de pior caso em tempo de projeto.

Em [BAK 91] são descritas extensões ao protocolo da prioridade máxima (PCP - "priority ceiling protocol"). Os autores apresentam as extensões em uma forma combinada, chamada de política de recurso baseada em pilha (SRP - "stack resource policy").

Em [JEF 92] é considerado o problema de escalonar um conjunto de tarefas esporádicas, cujo deadline é igual ao intervalo mínimo entre ativações. É suposto a existência de recursos que exigem exclusão mútua entre as tarefas durante o acesso. O modelo de tarefas inclui ainda tempos mínimo e máximo de execução para cada tarefa. É apresentado um algorítmo de escalonamento que combina o EDF com uma política de acesso aos recursos chamada DDM ("dynamic deadline modification"). É demonstrado um teste de escalonabilidade necessário e suficiente para o caso de todas as tarefas serem esporádicas. Este mesmo teste é suficiente mas não necessário quando existem tarefas periódicas.

2.4.2.3 Deadline Monotônico

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.

Modelo Básico

Em DM, prioridades são associadas às tarefas de forma inversamente proporcional ao tamanho do deadline (quanto menor o deadline, mais prioritária a tarefa). O DM é considerado um esquema de escalonamento ótimo [LEU 82], no sentido de que se o conjunto de tarefas apresenta um instante crítico e se o conjunto de tarefas não pode ser escalonado satisfatoriamente através do Deadline Monotônico, então não poderá ser escalonado através de nenhum método que utilize prioridades fixas. O conceito de instante crítico foi definido como o momento no qual todas as tarefas ficam prontas para executar simultaneamente.

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. As tarefas podem ser periódicas, habilitadas pela primeira vez no instante 0 (instante crítico) ou então tarefas esporádicas, com um intervalo de tempo mínimo entre habilitações. De qualquer forma, as tarefas são independentes. O único recurso a ser considerado é o processador.

Se o número de tarefas no sistema é N, cada tarefa recebe uma prioridade entre 1 (a tarefa mais prioritária) e N (a tarefa menos prioritária), conforme o DM. Neste texto, Ti identifica a tarefa com prioridade i, onde $1 \leq i \leq n$. Cada tarefa Ti é caracterizada em termos temporais pelo seu período Pi (tarefas periódicas), seu intervalo de tempo mínimo entre ativações Ii (tarefas esporádicas), seu tempo máximo de computação Ci e seu deadline Di. Estes valores são tais que:

- Para toda tarefa periódica Ti, $Ci \leq Di \leq Pi$ é verdadeiro;
- Para toda tarefa esporádica Ti, $Ci \leq Di \leq Ii$ é verdadeiro.

Para efeito de escalonamento, no pior caso, uma tarefa esporádica Ti comporta-se como uma tarefa periódica Ti', onde:

$$Di' = Di,\ Ci' = Ci \text{ e } Pi' = Ii.$$

Desta forma, tarefas esporádicas são tratadas naturalmente neste modelo. A figura 2.8 ilustra o comportamento temporal das tarefas neste modelo.

A partir do conceito de instante crítico, o artigo [AUD 90a] demonstra formalmente 4 testes de escalonabilidade. Estes mesmos testes são citados em [AUD 91a]. São eles:

- Um teste que verifica se o conjunto de tarefas é escalonável, suficiente mas não necessário, com complexidade $O(n)$;

- Um teste que verifica se o conjunto de tarefas é escalonável, suficiente mas não necessário, mais rigoroso que o anterior, com complexidade $O(n^2)$;
- Um teste que verifica se o conjunto de tarefas é não-escalonável, suficiente mas não necessário, com complexidade $O(n^2)$;
- Um teste que verifica se o conjunto de tarefas é escalonável, suficiente e necessário (exato), com complexidade pseudo-polinomial ([PAR 82a], [PAR 82b]).

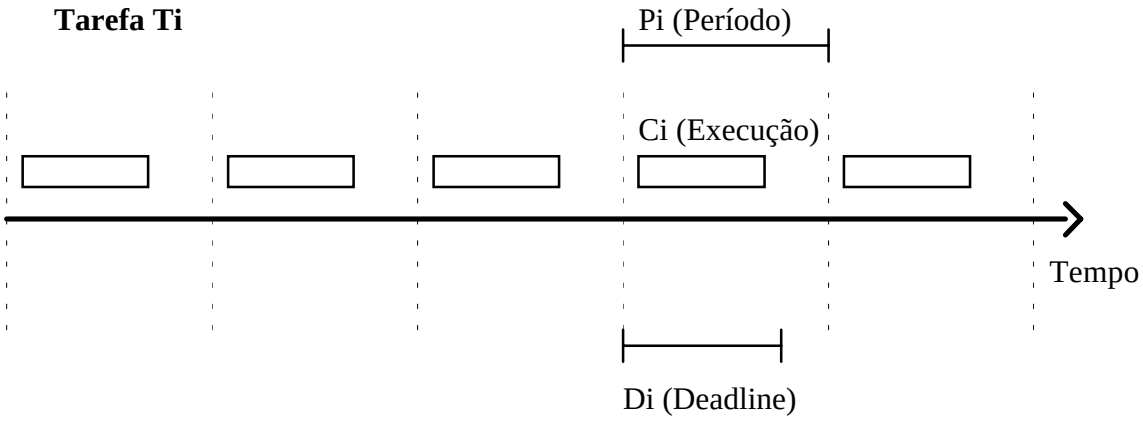

Figura 2.8 - Linha de tempo característica do Deadline Monotônico.

Em todos os testes é computada a interferência que as tarefas mais prioritárias causam sobre cada tarefa do conjunto. É verificado se, mesmo com tal interferência, cada tarefa do conjunto ainda consegue atender ao deadline. A lista acima ilustra um padrão. É relativamente comum encontrar testes de escalonabilidade suficientes mas não necessários com complexidade polinomial. Entretanto, testes exatos para modelos de tarefas além do trivial apresentam complexidade exponencial ou pseudo-polinomial.

Em [TIN 92a] é proposta uma solução para o ambiente distribuído. O artigo adota a solução usual, separando o problema de escalonamento em dois: a alocação de tarefas à processadores e o escalonamento local de cada processador. A alocação de tarefas à processadores é feita através do algorítmo subótimo conhecido por recozimento simulado ("simulated annealing"), descrito em [KIR 83]. Enquanto que algorítmos baseados em heurísticas exigem uma descrição implícita de "como construir uma solução", o recozimento simulado exige apenas uma função para "avaliar a qualidade de uma solução". O próprio algorítmo gera soluções, através de um caminhamento parcialmente aleatório através do espaço de soluções. Isto torna a técnica muito flexível. Em [TIN 92a], a alocação considera, além da escalonabilidade de cada processador, a quantidade de memória disponível e requisitos do tipo: duas tarefas não podem ser alocadas no mesmo processador ou duas tarefas devem ser alocadas no mesmo processador. O escalonamento local dos processadores é feito através do método DM. O modelo de tarefas supõe que tarefas podem enviar resultados para tarefas em outros processadores. Neste caso, é necessário considerar o atraso no meio de comunicação. O artigo considera um atraso máximo determinista, dependente do fluxo de dados na rede.

A figura 2.9 ilustra a situação da tarefa ***Ti***, que possui o deadline ***Di***. Se o atraso máximo no meio de comunicação for ***x***, então o deadline da tarefa é transformado da seguinte forma:

Di' = Di - x

O teste de escalonabilidade é aplicado considerando-se os deadlines transformados de todas as tarefas (que enviam mensagens através da rede). Se for garantido que a tarefa ***Ti*** sempre conclui sua execução dentro do deadline ***Di'***, então a mensagem com o resultado sempre chegará ao destino dentro do deadline ***Di***. Esta é uma técnica básica para obter soluções em ambiente distribuído, a partir de qualquer solução local que aceite tarefas com deadline menor que o período.

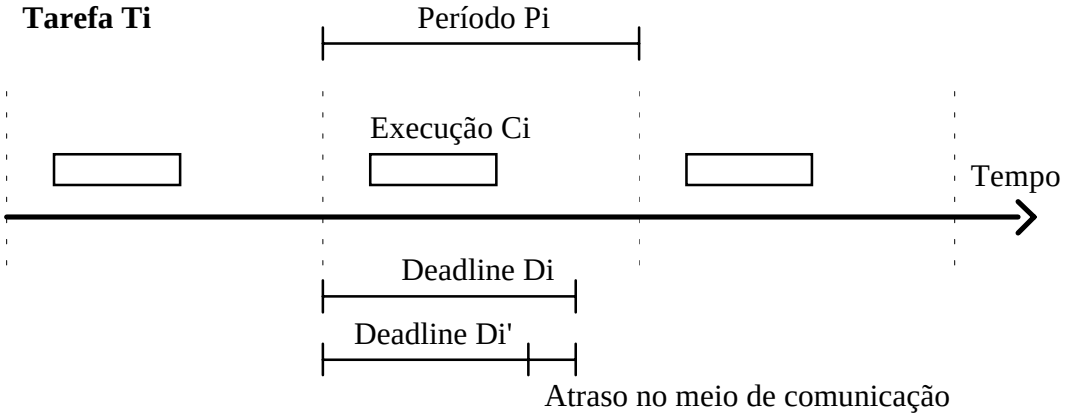

Figura 2.9 - Deadline Monotônico adaptado para ambiente distribuído.

Extensões ao Modelo Básico

Outros artigos ampliam o modelo de tarefas suportado pelo DM. Em [TIN 92b] é analisado o problema da mudança de modo operacional ("mode change") mantendo a garantia da escalonabilidade.

Em [AUD 93a] são descritas extensões possíveis a partir do modelo básico do método. O artigo mostra que diversas técnicas existentes para tratar o bloqueio de tarefas em função de exclusão mútua também podem ser usadas compostas com o DM. Em particular, são citadas herança de prioridade ("priority inheritance") [SHA 90], prioridade máxima ("priority ceiling") [SHA 90] e herança imediata de prioridade máxima ("immediate priority ceiling inheritance") [BAK 90]. O teste exato de escalonabilidade é adaptado para cada nova situação, mantendo a complexidade computacional pseudo-polinomial.

O teste de escalonabilidade usado em deadline monotônico verifica um conjunto de tarefas com prioridades fixas arbitrárias, não necessariamente atribuídas através do deadline monotônico. Assim, é possível que outros fatores, além do deadline, influenciem a definição das prioridades. Em [AUD 93a] é considerada a possibilidade de sobrecargas temporárias mesmo em sistemas com carga estática. O artigo propõe diversas técnicas para, manipulando a prioridade das tarefas, determinar em tempo de projeto quais tarefas

perderão o deadline em caso de sobrecarga. Nestes sistemas, a sobrecarga poderia ser causada por:

- Cálculo muito otimista do tempo de execução no pior caso;
- Custo ("overhead") do sistema maior do que o suposto;
- Tarefas esporádicas mais frequentes do que o suposto;
- Falhas no hardware, aumentando o tempo de execução das tarefas.

Em [LEH 92], Lehoczky e Ramos-Thuel apresentam um mecanismo para atender tarefas aperiódicas em sistemas que trabalham com escalonamento baseado em prioridades. O mecanismo determina o tempo máximo de processamento que pode ser tomado das tarefas periódicas garantidas, sem comprometer seus deadlines. O método é chamado de tomada de folga ("slack stealing"). O objetivo é minimizar o tempo de resposta das tarefas aperiódicas. É suposto que as tarefas periódicas foram garantidas através do emprego de deadline monotônico na definição de suas prioridades. O algorítmo proposto é ótimo no sentido de que ele conclui as tarefas aperiódicas no menor tempo possível, se comparado com qualquer outro algorítmo que também respeite os deadlines das tarefas periódicas.

A tomada de folga atende as tarefas aperiódicas usando, o mais cedo possível, qualquer tempo livre de processador que surja. Desta forma, qualquer folga que apareça na execução das tarefas garantidas é imediatamente usada para executar tarefas aperiódicas. O algorítmo apresentado em [LEH 92] constrói, em tempo de projeto, um mapa com o escalonamento previsto para um intervalo de tempo equiivalente ao MMC (mínimo múltiplo comum) dos períodos das tarefas garantidas. O mapa é então inspecionado para que a folga existente seja identificada. As folgas identificadas são armazenadas em uma tabela. Em tempo de execução, contadores são usados para manter a folga que pode ser tomada, a cada momento, em cada nível de prioridade. As tarefas aperiódicas são executadas seguindo uma disciplina do tipo "menor tempo de processamento restante primeiro".

Em [DAV 93a], a tomada de folga é adaptada para um modelo de tarefas que inclui sincronização entre tarefas, jitter na liberação e tarefas esporádicas garantidas. Em especial, é incluída na folga o tempo ganho quando uma tarefa não usa todo o tempo de processador que havia sido reservado para ela. Isto acontece com freqüência, pois a reserva é feita para o pior caso. Esta adaptação do algorítmo é possível na medida que o algorítmo proposto calcula as folgas em tempo de execução. É apresentado um algorítmo ótimo, porém com um custo computacional muito elevado. Os autores então desenvolvem uma família de algorítmos subótimos com um custo razoável. Em [DAV 93b] são também analisadas diversas aproximações para o algorítmo ótimo.

2.4.2.4 Análise Baseada em Períodos de Ocupação

A análise baseada em períodos de ocupação[3] (BP - "busy-periods analysis") permite encontrar o instante de conclusão de cada tarefa, no pior caso. 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

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

tempo de projeto. A análise BP parece admitir modelos de tarefas mais flexíveis que aqueles suportados atualmente pelos testes de escalonabilidade existentes para EDF, RM e DM.

A análise BP supõe que as tarefas possuem prioridades fixas e existe um escalonador preemptivo em tempo de execução. O critério para atribuir prioridades às tarefas não afeta a análise, que parte das tarefas já com suas respectivas prioridades definidas.

Esta seção apresenta como modelo básico a proposta que aparece em [TIN 94a]. Entretanto, este tipo de análise apareceu antes em [LEH 90], para um modelo de tarefas mais simples.

Modelo Básico

Em [TIN 94a] é apresentada uma proposta que utiliza a análise BP, a partir de prioridades fixas e escalonador preemptivo em tempo de execução. O artigo supõe conjuntos de tarefas periódicas ou esporádicas com deadlines arbitrários, ou seja, o deadline pode ser menor, igual ou maior do que o período (periódicas) ou do que o intervalo mínimo entre ativações (esporádicas). As tarefas podem sofrer jitter na liberação devido ao disparo por um escalonador dirigido por ticks. Também considera tarefas esporádicas do tipo rajada ("bursty"), ou seja, tarefas chegam esporadicamente mas então executam periodicamente por um tempo limitado. A figura 2.10 ilustra uma tarefa tipo rajada.

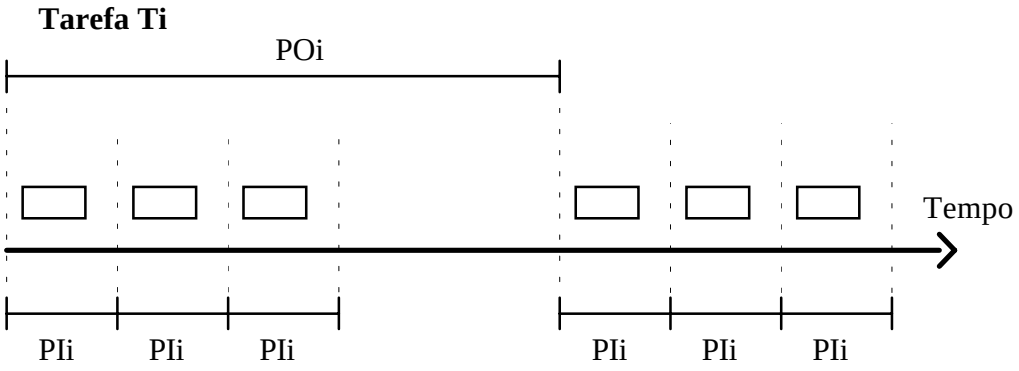

Figura 2.10 - Linha de tempo de uma tarefa tipo rajada ("bursty").

O modelo de tarefas considerado em [TIN 94a] inclui um conjunto de ***N*** tarefas executando em um monoprocessador. Cada tarefa ***Ti*** possui uma prioridade fixa ***i***, onde ***i*** está entre ***1*** e ***N***, um tempo máximo de computação ***Ci*** e um deadline qualquer ***Di***. Cada tarefa ***Ti*** ainda apresenta um jitter na liberação ***Ji*** ("release jitter"). O acesso exclusivo à recursos pode ser implementado através do protocolo da prioridade máxima ("priority ceiling protocol", [SHA 90]) ou através da política de recurso baseada em pilha ("stack resource policy", [BAK 91]). Cada tarefa ***Ti*** pode ser uma tarefa periódica com período ***Pi***, uma tarefa esporádica com intervalo mínimo entre ativações ***Ii*** ou ainda uma tarefa tipo rajada, com um tempo mínimo entre rajadas ***POi*** ("outer period") e um tempo mínimo entre ativações dentro da rajada ***PIi*** ("inner period"). É importante salientar que não existem restrições com respeito ao valor do deadlline ***Di***.

A proposta em [TIN 94a] emprega uma análise baseada em períodos de ocupação (BP - "busy-periods analysis") para encontrar o instante de conclusão de cada tarefa, no pior caso. Se o instante de conclusão, no pior caso, for menor que o deadline da tarefa, significa que o deadline será sempre cumprido.

Esta análise é construida em torno do conceito de períodos de ocupação ("busy periods"). Um período de ocupação de nível ***i*** corresponde a um intervalo de tempo onde são executadas, continuadamente, tarefas com prioridade igual a ***i*** ou superior. É mostrado que o instante de conclusão no pior caso de uma tarefa ***Ti***, com prioridade ***i***, pode ser encontrado através do exame de uma série de janelas na linha de tempo. Cada janela inicia com uma ativação da tarefa ***Ti*** e termina exatamente no final do respectivo período de ocupação ***i***. A partir de uma ativação qualquer da tarefa ***Ti***, escolhida arbitrariamente, é necessário analisar um determinado número de janelas para o passado. Este número depende dos parâmetros do problema. Em particular, da relação entre ***Di*** e ***Pi*** (tarefa periódica), ***Di*** e ***Ii*** (tarefa esporádica) ou ***Di***, ***POi*** e ***PIi*** (tarefa tipo rajada). Esta análise ocorre em tempo de projeto. O resultado final é obtido através de um método iterativo, garantidamente finito.

Um teste de escalonabilidade construido a partir da análise BP é exato. Entretanto, o método iterativo empregado pode ser custoso em termos computacionais, para determinados dados de entrada. É possível construir um teste de escalonabilidade suficiente mas não necessário introduzindo limites de tempo no método iterativo.

Extensões ao Modelo Básico

Nos modelos de tarefas descritos antes, foi sempre suposta a existência de um instante no tempo quando todas as tarefas são liberadas simultaneamente, ou seja, o instante crítico definido em [LIU 73]. Esta suposição nem sempre corresponde à realidade. Em uma aplicação tempo real podem haver tarefas com o mesmo período, porém deslocadas no tempo. Desta forma, estas tarefas jamais serão liberadas simultaneamente. A suposição do instante crítico torna o escalonamento do conjunto mais difícil do que ele realmente é.

A figura 2.11 ilustra a existência de um deslocamento no tempo ***Oi***, associado com a tarefa ***Ti***. Supondo que o sistema iniciou sua execução no instante ***0***, cada período da tarefa ***Ti*** inicia no instante ***x.Pi*** e termina no instante ***(x+1).Pi***, onde ***x = 0,1,2,...*** enumera as sucessivas habilitações da tarefa ***Ti***. A cada período, a tarefa somente pode iniciar sua execução a partir do instante ***x.Pi+Oi***.

Em [AUD 93b], o modelo de tarefas associado com BP é extendido para permitir que tarefas possuam deslocamento entre si. Ainda em [AUD 93b] são fornecidas indicações de como a possibilidade de deslocamentos no modelo de tarefas permite a construção de aplicações distribuídas. O artigo mostra que BP é bastante extensível e que conjuntos de tarefas, antes somente escalonáveis através de executivo cíclico, podem agora ser implementados com prioridades fixas através desta proposta.

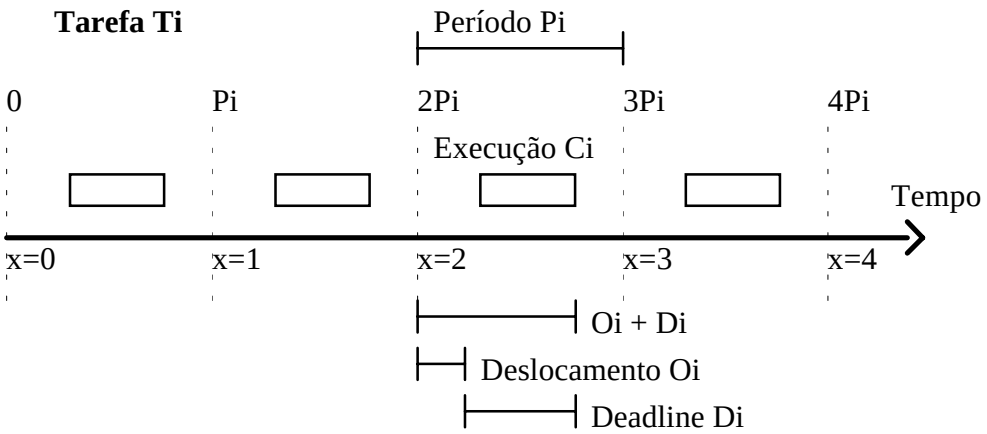

Figura 2.11 - Deslocamento ("offset") de uma tarefa em relação ao início do período.

A proposta BP é bastante promissora. Apesar de recente, já existem diversos trabalhos que procuram extender o seu modelo de tarefas. Em [BUR 94] o modelo de tarefas é extendido para permitir que os deadlines sejam associados com o último evento observável gerado por uma tarefa, e não o término da sua computação. Em [DAV 93a] é mostrado que o algorítmo para tomada de folga usado com DM pode também ser usado com BP. Uma descrição completa da proposta BP pode ser encontrada na tese [TIN 94b].

Atribuição de Prioridades às Tarefas

É importante observar que esta análise independe do critério utilizado para atribuir prioridades às tarefas. A análise apenas determina se, com a atribuição de prioridades escolhida, é possível ou não garantir o cumprimento de todos os deadlines.

Em [LEH 90] é mostrado que RM e DM não são ótimos para atribuir prioridades quando as tarefas possuem deadlines arbitrários, ou seja, deadlines que podem ser menor, igual ou maior que o período da tarefa. Em [AUD 91b] é apresentado um algorítmo que faz uma atribuição de prioridades ótima. Ótima no sentido que se este algorítmo não consegue uma ordem de prioridades onde todas as tarefas são escalonáveis, então não existe uma ordem de prioridades onde todas as tarefas são escalonáveis. O emprego deste algorítmo é sugerido em [TIN 94a].

O objetivo do algorítmo descrito em [AUD 91b] é ordenar as tarefas de sorte que a posição de uma tarefa nesta ordem reflita sua prioridade. O algorítmo trabalha com as tarefas divididas em dois grupos. Um grupo já ordenado, contendo as tarefas com as prioridades inferiores já definidas e um grupo desordenado, com as tarefas que possuirão as prioridades superiores. Este algorítmo tem uma complexidade ***O((N2 + N) . E)***. O valor ***E*** representa a complexidade do teste de escalonabilidade.

2.4.2.5 Outras Propostas

O número de propostas existente na bibliografia que trabalham com prioridades e teste de escalonabilidade é muito grande. Em [BUR 93] é proposto o uso de prioridades em um sistema distribuído sobre uma rede de computadores com arquitetura ponto-a-ponto. As

tarefas são alocadas estaticamente aos processadores através do algorítmo recozimento simulado ("simulated annealing"). É possível especificar restrições na alocação do tipo "réplicas devem ficar em processadores diferentes". Este mesmo algorítmo é utilizado para atribuir prioridades às tarefas. Como os deadlines são associadas às atividades, o próprio algorítmo particiona o deadline da atividade, gerando assim deadlines para tarefas. Outras propostas recentes dentro desta classe podem ser encontradas em [HAR 94], [HA 94], [HOM 94] e [TIA 94].

2.4.3 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 algorítmos propostos em [RAM 89], [CHE 90a] e [ELH 94]. Entretanto, os algorítmos pertencentes a esta abordagem tratam basicamente do escalonamento de tarefas não garantidas em tempo de projeto. São algorítmos que descartam a tarefa caso não seja possível cumprir seu deadline. Após o deadline, o valor da tarefa para o sistema cai a zero.

Exemplo de Proposta para Monoprocessadores

Em [SCH 92] é apresentado um algorítmo de escalonamento dentro desta abordagem. O modelo de tarefas considera um conjunto de ***N*** tarefas executando em um monoprocessador. As tarefas podem ser preemptadas. Também é possível a existência de relações de precedência entre as tarefas. Cada tarefa ***Ti***, com ***i*** entre ***1*** e ***N***, é caracterizada por um tempo de liberação ***Li*** ("arrival"), um tempo de pronto ***Ri*** ("start time"), um tempo máximo de computação ***Ci*** ("maximum computation time") e um deadline ***Di***.

O algorítmo apresentado é baseado no EDF ("earliest deadline first"). Ele seleciona a próxima tarefa que vai executar, no momento de uma troca de contexto. Ao mesmo tempo, é usado para aceitar ou rejeitar uma tarefa no momento da sua liberação, realizando um teste de escalonabilidade dinâmico. Este teste verifica se a nova tarefa pode ser escalonada de forma a atender suas restrições temporais e de precedência. Como é a regra nesta abordagem, [SCH 92] assume que "qualquer rejeição de tarefa resultante são tratadas pelos níveis mais altos do sistema operacional de tempo real".

O escalonamento se dá com preempção e o algorítmo apresenta complexidade computacional ***O(N.logN)***. Ele é ótimo no sentido de que se ele não conseguir garantir o deadline de uma tarefa que chega, nenhum outro algorítmo nas mesmas condições consegue. O algorítmo apresentado não considera explicitamente as relações de precedência entre as tarefas. Entretanto, tais restrições podem ser impostas através da manipulação dos momentos de pronto e das deadlines das tarefas. O artigo descreve uma extensão ao algorítmo que faz isto.

Os autores ainda mostram que o algorítmo é capaz de lidar com deadlines associadas à atividades e não à tarefas individuais. Neste caso, todas as tarefas da atividade devem ser aceitas ou todas devem ser rejeitadas. A aceitação ou rejeição da atividade completa é feita com uma complexidade computacional ***O((M + K) . Log (M + K))***, onde ***M*** é o número de tarefas pertencentes a atividade e ***K*** é o número de tarefas (excluindo as tarefas da atividade em questão) envolvidas no reescalonamento realizado pelo algorítmo.

Teoria das filas é aplicada para mostrar que a complexidade do caso médio do algorítmo é muito melhor que a complexidade no pior caso. Os autores também relatam em [SCH 92] alguns resultados experimentais, os quais confirmam os resultados analíticos obtidos.

A literatura referente a esta classe de propostas é bastante extensa. Outras propostas interessantes dentro desta abordagem, para monoprocessadores, podem ser encontradas em [CHE 90a] e [KOR 92]. Em [ELH 94] o algorítmo deadline monotônico é adaptado para uso dentro desta abordagem, também em monoprocessadores.

Exemplo de Proposta para Multiprocessadores

Em [RAM 90] são apresentados dois algorítmos, baseados em heurísticas, para escalonar conjuntos de ***N*** tarefas em multiprocessadores. As tarefas são caracterizadas por deadlines e uso de recursos além dos processadores (necessidade de prover exclusão mútua entre tarefas). O modelo de tarefas supõe tarefas que não possuem relações de precedência entre si e que não podem sofrer preempção.

No modelo de tarefas usado, uma tarefa ***Ti*** é caracterizada por um momento de liberação ***Li*** ("arrival time"), um deadline ***Di***, um tempo máximo de computação ***Ci*** ("worst case processing time") e pelos recursos necessários ***Qi*** ("resource requirements"). Uma tarefa pode usar um recurso em modo compartilhado ou em modo exclusivo. Todo recurso solicitado é mantido pela tarefa durante toda a sua execução.

O primeiro algorítmo, chamado de algorítmo original, é uma extensão de um algorítmo apresentado antes em [ZHA 87] para monoprocessadores. Ele possui complexidade computacional $O(N^2)$. O segundo algorítmo, chamado de algorítmo míope, é uma versão simplificada do anterior, com complexidade computacional ***O(N)***. Os algorítmos foram avaliados através de simulação. Estes algorítmos foram criados com a intensão de serem usados no sistema Spring ([STA 87], [STA 89], [STA 91]), para fazer o escalonamento local em nodos que são multiprocessadores.

Escalonar um conjunto de tarefas, de maneira que seus requisitos temporais sejam atendidos, pode ser visto como um problema de pesquisa em árvore. A raiz representa uma

escala de execução vazia. Acrescentar uma tarefa à esta escala significa descer um nível na árvore. Um caminho da raiz até uma folha representa uma escala de execução que inclui todas as tarefas, embora não necessariamente satisfazendo seus requisitos. O objetivo da pesquisa é encontrar um caminho que satisfaça todos os requisitos das tarefas.

Uma pesquisa exaustiva sobre esta árvore é proibitiva em termos de esforço computacional. Por isto, heurísticas são empregadas em uma pesquisa subótima, onde nem todos os caminhos possíveis da raiz às folhas são analisados. A pesquisa parte da raiz da árvore. A cada passo da pesquisa, uma função heurística ***H*** decide qual será a próxima tarefa a ser incluida na escala de execução. A seguir, a escala de execução parcial obtida até o momento é testada para determinar se este caminho ainda é promissor. Em outras palavras, este determina se a pesquisa deve proceguir a partir deste ponto, ou se um outro caminho deve ser tentado ("backtracking"). Em [RAM 90] são apresentadas diversas funções ***H*** que podem ser aplicadas. Entre elas estão:

- Tenta escalonar antes as tarefas com deadline mais próxima;
- Tenta escalonar antes as tarefas com menor tempo de computação;
- Tenta escalonar antes as tarefas que podem iniciar mais cedo suas respectivas execuções, considerando as necessidades de recursos;
- Tenta escalonar antes as tarefas com a menor folga, considerando as necessidades de recursos;
- Combinações dos critérios acima.

Em função de resultados obtidos através de simulações, os autores optam por uma função do tipo $H(i) = Ci + W \cdot Si$, onde ***W*** é uma constante arbitrada e ***Si*** é o primeiro instante que ***Ti*** poderá iniciar sua execução, em função dos recursos necessários. Os algorítmos tentam escalonar antes a tarefa ***Ti*** com o menor valor de ***H(i)***.

Os dois algorítmos apresentados em [RAM 90] diferem com respeito ao conjunto de tarefas considerado no momento de aplicar a função H e estender uma solução parcial. O algorítmo original como candidatas todas as tarefas que ainda não foram escalonadas. O algorítmo míope restringe sua atenção a um pequeno subconjunto de tarefas ainda não escalonadas: aquelas com os deadlines mais próximos.

Os dois algorítmos de escalonamento são avaliados através de simulação. Para cada um deles foi computado o percentual de vezes na qual um conjunto de tarefas, sabidamente escalonável, foi efetivamente escalonado com sucesso. Os autores mostram em [RAM 90], através dos resultados das simulações, que:

- Para um dado custo máximo de escalonamento, o algorítmo míope trabalha tão bem quanto o algorítmo original, nos casos em que as tarefas possuem deadlines apertadas ou os conflitos por recursos são muito significativos;
- Para um dado custo máximo de escalonamento, o algorítmo míope pode trabalhar melhor que o algorítmo original, nos casos em que as tarefas possuem deadlines folgados ou os conflitos por recursos são pouco significativos;
- Em geral, o algorítmo míope apresenta um custo computacional consideravelmente menor que o algorítmo original, um aspecto importante em escalonamento dinâmico.

Exemplo de Proposta para Ambiente Distribuído

Em [RAM 89], Ramamritham, Stankovic e Zhao descrevem um conjunto de algorítmos baseados em heurísticas para escalonar tarefas em ambiente distribuído. Eles também são empregados no sistema Spring. O modelo de tarefas empregado inclui tarefas com deadlines e necessidade de recursos além do processador. A carga é suposta dinâmica.

Quando uma tarefa é ativada em um nodo qualquer da rede, o escalonador local tenta garantir sua execução neste mesmo nodo, dentro do deadline solicitado, através de um teste de escalonabilidade. Quando isto não é possível, os escalonadores em diferentes nodos cooperam no sentido de localizar um nodo onde a tarefa poderá ser executada dentro do deadline solicitado. Como é característico desta abordagem, as vezes a tarefa é rejeitada. Isto acontece quando nenhum nodo da rede tem condições de executa-la dentro do deadline solicitado.

Em [RAM 89] são apresentadas quatro soluções para o problema de garantir uma tarefa recém chegada ao sistema. É suposto que as quatro soluções avaliadas usam o mesmo algorítmo de escalonamento local, descritos em [ZHA 87] ou [RAM 90]. Elas diferem com respeito a heurística empregada no momento de localizar um nodo para enviar tarefas que não podem ser escalonadas localmente. Em resumo, as quatro propostas são:

- Escalonamento aleatório ("random scheduling"), envia a tarefa para um nodo selecionado de maneira aleatória.
- Endereçamento focado ("focused addressing"), envia a tarefa para um nodo que é suposto com capacidade no momento para executar a tarefa e atender ao deadline.
- Leilão ("bidding"), envia a tarefa para o nodo que tiver enviado a melhor proposta para o nodo origem. Este algorítmo foi originalmente apresentado em [RAM 84] e a sua estabilidade foi analisada em [STA 85a].
- Algorítmo flexível ("flexible"), combina as técnicas empregadas no endereçamento focado e no leilão. Este algorítmo foi originalmente apresentado em [STA 85b].

No modelo de tarefas descrito em [RAM 89], existem ***M*** nodos em um sistema distribuido fracamente acoplado. Cada nodo contém um conjunto de ***X*** recursos distintos. Um recurso aqui é uma abstração que pode representar processador, dispositivo de entrada ou saída, arquivo, estrutura de dado, etc. Um recurso ativo possui poder de processamento, enquanto os demais são recursos passivos. Cada tarefa necessita, no mínimo, de um recurso ativo. Alguns recursos podem ser usados simultaneamente por várias tarefas, enquanto possuem características que exigem um acesso exclusivo. As tarefas são a unidade de escalonamento, ou seja, tarefas não podem ser preemptadas. Cada tarefa ***Ti*** é caracterizada por um tempo máximo de computação ***Ci*** ("worst case computation time"), um deadline ***Di*** e a necessidade de recursos da tarefa.

As quatro propostas são avaliadas através de simulação e comparadas com dois algorítmos de referência. Na referência pior, se não consegue escalonar localmente, não envia a tarefa para nenhum outro nodo. Na referência melhor, cada nodo possui uma visão perfeitamente atualizada do estado de toda a rede. A métrica usada para comparar os diferentes algorítmos é a taxa de garantia ("guarantee ratio"), ou seja, o número de tarefas que puderam ser executadas e ter seus deadlines atendidos dividido pelo número total de tarefas ativadas.

As simulações descritas em [RAM 89] comparam as quatro heurísticas em diferentes situações de carga, diferentes atrasos na comunicação, diferentes níveis de folga e diferentes distribuições quanto a ativação de tarefas. As simulações mostram que o algorítmo flexível apresenta um comportamento ligeiramente melhor que os demais. Bastante interessante é o fato do algorítmo aleatório apresentar um desempenho muito próximo ao algorítmo flexível.

2.4.4 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 descritas nesta seção 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).

Nos modelos mais tradicionais, o valor de uma tarefa para o sistema não depende do instante da conclusão, mas sim do cumprimento ou não do deadline. Na proposta descrita em [JEN 85], não existe propriamente um deadline. Cada tarefa possui uma função benefício na forma de uma curva "valor para o sistema" versus "instante de conclusão da tarefa" ("time-value function"). O escalonamento é feito no sentido de maximizar as constribuições das tarefas para o sistema. É suposto um ambiente onde tarefas frequentemente são liberadas, em intervalos irregulares, e possuem tempos de computação aleatórios.

O modelo de tarefas inclui um conjunto de ***N*** tarefas preemptáveis, residentes em um computador com uma única memória compartilhada e um ou mais processadores. As tarefas são independentes entre si, ou seja, não existem relações de precedência ou de exclusão mútua. Cada tarefa ***Ti*** é caracterizada por um momento de pronto ***Ri***, um tempo de computação estimado ***Ci*** e uma função benefício ***Vi(t)***.

A função ***Vi(t)*** define o valor do benefício que representa para o sistema concluir a tarefa ***Ti*** no instante ***t***. Existem duas fontes possíveis para a função ***Vi(t)***. Uma é o implementador da tarefa, a partir dos requisitos internos do sistema que afetam aquela tarefa. Outra fonte é o arquiteto do sistema, o qual está ciente do relacionamento entre aquela tarefa e o sistema como um todo, inclusive o ambiente externo.

A introdução do conceito de função benefício ***Vi(t)*** não exclui a existência da noção de um deadline para a tarefa. É possível construir uma função benefício com uma descontinuidade que represente o conceito usual de deadline. Isto é ilustrado na figura 2.12. Entretanto, também é possível construir uma função benefício onde não é possível identificar um deadline, como o termo é empregado normalmente. A figura 2.13 ilustra esta outra situacão. Fica claro que o conceito de função benefício é mais amplo que o conceito de deadline e permite representar um leque mais amplo de situações práticas, como mostra

a figura 2.14. Uma aplicação pode, inclusive, possuir uma variada mistura de tarefas cujas funções benefício correspondem a curvas bastante diferentes.

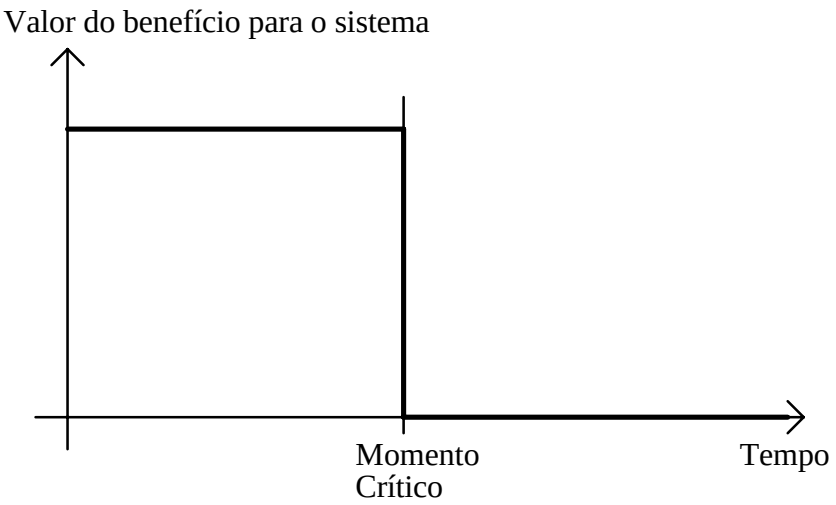

Figura 2.12 - Função benefício onde existe um deadline tradicional.

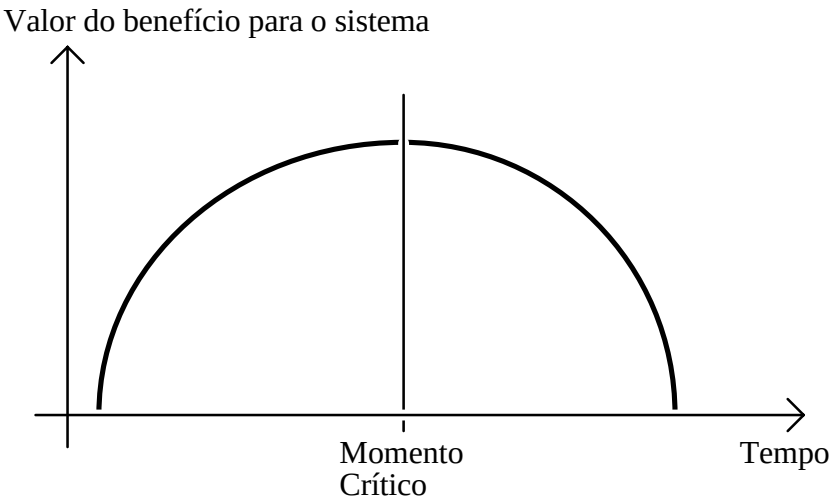

Figura 2.13 - Função benefício onde não existe um deadline tradicional.

Em [JEN 85] é usada simulação para avaliar diversos algorítmos e comparar seus desempenhos relativos. As funções benefício das tarefas usadas na simulação não possuem uma forma qualquer. Elas foram limitadas de maneira facilitassem o tratamento do problema, ainda conservando um alto poder de expressão. As funções empregadas são formadas por duas partes. A primeira parte vai até o chamado momento crítico ("critical time"). A segunda parte inicia neste mesmo momento crítico. O fato das duas partes se encontrarem no momento crítico permite que ali exista uma descontinuidade na função benefício, se a aplicação assim requerer. O conceito de momento crítico usado aqui não

possui nenhuma relação com o conceito de instante crítico definido em [LIU 73] e apresentado no início deste capítulo.

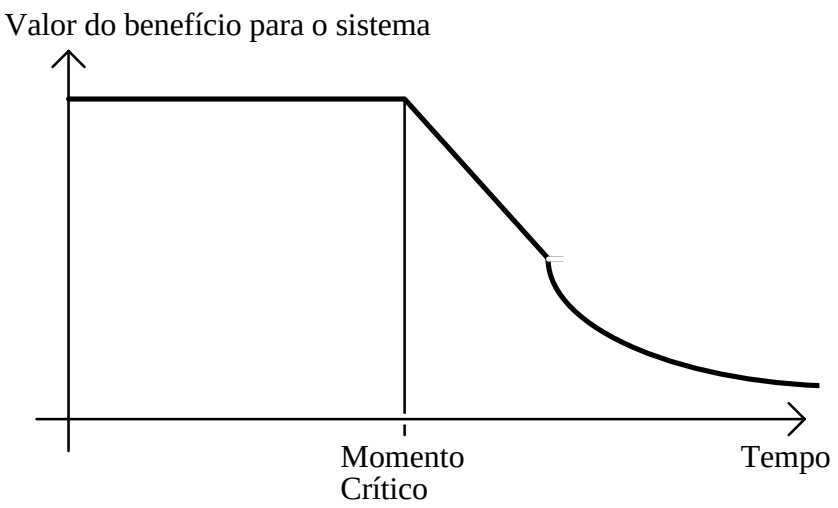

Figura 2.14 - Exemplo da flexibilidade introduzida através do conceito de função benefício.

Para cada uma das duas partes da função benefício, cinco constantes são usadas para definir a função em relação ao momento crítico. A função é definida pela equação:

$$Vf(t) = K_1 + K_2 . t - K_3 . t^2 + K_4 . e - (K_5 . t)$$.

O simulador usa um modelo estatistico de carga tempo real para gerar conjuntos de tarefas, periódicas e aperiódicas. Foram empregados quatro formatos diferentes para as funções benefício na geração da carga. Durante a simulação, são computados diversos valores:

- Valor total do sistema, ou seja, benefício total obtido pelo sistema;
- Número médio de preempções, ou seja, número médio de vezes que uma tarefa executando perde o processador;
- Atraso médio, ou seja, diferença média entre o momento da conclusão da tarefa e o seu momento crítico;
- Atraso máximo, ou seja, a maior diferença entre o momento da conclusão de uma tarefa e seu respectivo momento crítico;
- Média dos atrasos não negativos, ou seja, média dos atrasos considerando apenas tarefas que foram concluidas depois do momento crítico;
- Número de tarefas tardias, ou seja, número total de tarefas concluidas após o momento crítico.

Foram usados na simulação diversos algorítmos de escalonamento clássicos, tais como:

- Tarefa com menor tempo de execução executa antes;
- Tarefa com momento crítico mais cedo é executada antes;

- Tarefa com menor folga[4] ("slack time") é executada antes;
- Tarefa que foi liberada antes executa primeiro ("FIFO");
- Tarefa que vai executar a seguir é escolhida randomicamente;
- Cada tarefa recebe uma prioridade fixa proporcional ao valor máximo da sua função benefício, sempre executa antes a tarefa com maior prioridade.

Além dos algorítmos citados acima, os autores propõe dois novos algorítmos, mais adequados ao modelo de tarefas proposto. O algorítmo BEValue1 escolhe para executar antes as tarefas com maior densidade de valor ("value density"), a qual é calculada por:

(valor obtido na conclusão da tarefa) / (tempo de computação restante).

O algorítmo BEValue2 inicia colocando as tarefas na ordem crescente de seus momentos críticos. A seguir, é verificada a possibilidade de cada tarefa ser concluida antes do seu momento crítico. Quando o número de tarefas cuja conclusão será após o momento crítico ultrapassa um determinado valor, o sistema é considerado em sobrecarga ("overload"). Neste caso, a tarefa com menor densidade de valor é removida. Esta operação é repetida até que a sobrecarga seja eliminada.

O melhor desempenho entre todos os algorítmos simulados é obtido com o algorítmo BEValue2. Ele supera todos os outros algorítmos por margens significativas e mostra uma grande consistência de desempenho. O algorítmo BEValue1 apresentou um comportamente muito variável, dependente dos valores das funções usadas na carga. Por ser um algorítmo ganancioso, preferindo pegar qualquer valor antes em vez de esperar para pegar um valor maior depois, ele perde muitas oportunidades de concluir tarefas antes do momento crítico.

Existem na bibliografia outros trabalhos que seguem também esta abordagem. Os próximos parágrafos citam alguns deles. Em [WEN 88] é descrita uma implementação experimental de um algorítmo semelhante aos descritos em [JEN 85], com uma complexidade computacional $O(N^2)$. O maior problema desta proposta é o alto custo do escalonamento. Em um monoprocessador com carga elevada mais de 80% do tempo do processador é gasto com o algorítmo de escalonamento. A implementação descrita utiliza um processador auxiliar, dedicado exclusivamente à execução do algorítmo de escalonamento.

Em [CHE 91] é seguida a mesma abordagem dos artigos citados antes. O autor propõe um algorítmo subótimo com complexidade polinomial para minimizar a penalidade imposta ao sistema pelo atraso de tarefas. O artigo considera que cada tarefa possui um momento de conclusão ótimo ("optimal completion time"). O valor do benefício gerado por uma tarefa se reduz de maneira quadrática na medida que o instante real de conclusão se afasta do instante ótimo. Em outras palavras, é suposto que as curvas que representam funções benefício possuem sempre o mesmo formato, representado pela redução quadrática do benefício em função do tempo. O comportamento do algorítmo é avaliado através de simulacões.

---

[4]A folga é calculada como:

(momento crítico) - (momento atual) - (tempo de computação que falta para concluir a tarefa)

Em [KAO 94] é analisado o problema do particionamento do deadline de uma atividade em deadlines parciais para as tarefas que compõe a atividade. É estudada a situação na qual uma atividade é composta por diversas tarefas em paralelo, diversas tarefas em série e uma combinação das duas situações.

Em [CHE 94] é mostrado como políticas empregadas em sistemas que descartam tarefas podem ser mapeadas em políticas que atrasam tarefas ao invés de descarta-las. Neste trabalho, uma política é considerada melhor que outra se resultar em uma menor taxa de perda de deadlines (TPD). A TPD é calculada por:

(Número de tarefas que perderam o prazo) / (Número total de tarefas invocadas).

Os autores mostram que diversas políticas ótimas para sistemas que descartam tarefas também são ótimas nesta abordagem. Os resultados são ilustrados através de uma aplicação: um subsistema de acesso à disco.

2.4.5 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 algorítmos 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 tratada em detalhes no capítulo 3.

2.5 Comparação entre Abordagens

Nesta seção é apresentada uma comparação entre as diferentes abordagens citadas na seção 2.3 e exemplificadas na seção 2.4 desde capítulo. Obviamente, cada autor em particular destaca as vantagens da abordagem que ele segue.

2.5.1 Comparação entre Abordagens que Oferecem Garantia

Existe na literatura uma discussão sobre qual a melhor abordagem entre aquelas que oferecem previsibilidade determinista. Os principais argumentos a favor de uma e de outra abordagem podem ser encontrados em [LOC 92], [XU 93a] e [TIN 94b].

A favor do executivo cíclico, são normalmente citadas as seguintes vantagens:

- Na maioria das aplicações atendidas por STR-H, a maior parte da computação é realizada por tarefas periódicas, que simulam um comportamento contínuo. Os eventos esporádicos são menos freqüentes e podem ser tratados por servidores periódicos. Em outras palavras, o modelo de tarefas normalmente associado com executivos cíclicos é apropriado para as tarefas encontradas na prática.

- Embora muitos dos algorítmos empregados no cálculo da grade de execução possuam complexidade exponencial, as instâncias de problemas encontradas na prática são tratáveis em tempos satisfatórios. Como o cálculo da grade é feito em tempo de projeto, seu tempo de execução possui uma importância limitada.

- O executivo cíclico oferece a vantagem de um menor custo em tempo de execução, pois todas as decisões são tomadas em tempo de projeto e o número de chaveamentos de contexto pode ser minimizado.

- Geralmente, os modelos de tarefas utilizados pela abordagem baseada em prioridades exigem preempção. Estes modelos não comportam tarefas que não podem ser preemptadas durante a execução. Tal restrição não existe no executivo cíclico.

- Através da atribuição de prioridades às tarefas é possível obter-se apenas um subconjunto das possíveis seqüências de execução para um dado conjunto de tarefas. Por exemplo, na presença de recursos, as vezes a melhor solução implica em deixar o processador livre até a liberação de uma determinada tarefa, mesmo com outras tarefas prontas para executar. Isto jamais acontece quando escalonadores dirigidos por prioridade são utilizados. Entretando, uma grade de execução pode perfeitamente conter esta solução.

- Na abordagem baseada em prioridades, são empregados mecanismos clássicos de sincronização, tais como semáforos e monitores. Estes mecanismos implicam em um custo durante a execução, além de bloqueios que, no pior caso, podem ser grandes. No executivo cíclico, a sincronização das tarefas é obtida na própria construção da grade.

- É fácil obter um jitter baixo. Basta levar este critério em consideração no momento de construir a grade de execução. O controle do jitter é muito importante em sistemas de controle por realimentação. A estabilidade da função de transferência básica depende do controle rígido do jitter no laço de realimentação.

A favor da abordagem baseada em teste de escalonabilidade e prioridades, são citados as seguintes vantagens:

- Nos sistemas baseados em prioridades não é possível saber, em tempo de projeto, exatamente qual tarefa vai executar quando. Entretanto, estes sistemas atendem ao requisito básico da previsibilidade temporal ao garantirem, em tempo de projeto, que todos os deadlines serão cumpridos. O conhecimento antecipado completo do comportamento temporal, como provido pelo executivo cíclico, não é realmente necessário. O necessário é garantir o atendimento dos requisitos temporais, na forma de deadlines.

- Aplicações construidas através do executivo cíclico são inerentemente frágeis. Um erro qualquer no projeto do software (por exemplo, tempo máximo de execução substimado) resulta em um comportamento imprevisível do sistema. No caso das prioridades, é possível associar prioridades mais altas com as tarefas mais importantes, obtendo uma degradação controlada no caso de faltas que resultem em sobrecargas temporárias.

- Para que a grade usada pelo executivo cíclico não fique demasiadamente grande, existe a tendência de fazer com que o período de todas as tarefas sejam harmônicos entre si. Desta

forma, o MMC será igual ao período da tarefa mais lenta. Isto é conseguido através de mudanças na especificação do sistema, normalmente resultando em maior gasto de recursos (processador) ou menor qualidade do sistema. Este problema não existe quando prioridades são utilizadas.

- O executivo cíclico é criticado por não ser capaz de lidar bem com tarefas esporádicas. Como o momento no qual a tarefa esporádica será ativada não é conhecido em tempo de projeto, não é possível incluí-la na grade de execução. O emprego de servidores periódicos de tarefas esporádicas pode resultar em grande desperdício do tempo do processador.

Apesar desta discussão estar presente na literatura, existe atualmente uma certa equivalência entre as duas abordagens. Os executivos cíclicos levam vantagem no aspecto de adaptarem-se mais facilmente à novas propriedades no modelo de tarefas. O executivo cíclico também oferece um custo menor em tempo de execução, uma vez que todo o escalonamento é calculado em tempo de projeto. Entretanto, o escalonamento baseado em prioridades é mais flexível para lidar com tarefas esporádicas garantidas ou aperiódicas não garantidas. As duas abordagens são normalmente obrigadas a lidar com problemas de complexidade exponencial em tempo de projeto, associados com o cálculo da grade de execução e com o teste de escalonabilidade.

Novos trabalhos estão sendo publicados em maior número dentro da abordagem baseada em prioridades. Eles procuram flexibilizar o modelo de tarefas suportado, provendo novos testes de escalonabilidade.

2.5.2 Previsibilidade, Adaptabilidade e Utilização de Recursos

Examinando a literatura da área, é possível constatar que existem dois dilemas básicos em sistemas tipo tempo real: "garantia versus utilização dos recursos" e "garantia versus adaptabilidade". Estas duas dicotomias tem origem nas condições necessárias para que seja obtida uma previbilidade determinista em tempo de projeto, ou seja:

- Que a carga seja limitada e conhecida em tempo de projeto;
- Que recursos sejam reservados para a execução de todas as tarefas no pior caso.

Para que os deadlines das tarefas possam ser garantidos em tempo de projeto, é necessário o conhecimento, à priori, da demanda por recursos no pior caso. Também é necessária a disponibilidade de recursos para esta demanda no pior caso. Tais exigências não são pequenas. Em um sistema grande, os recursos necessários para a demanda no pior caso podem ser proibitivos.

Por outro lado, adaptabilidade se traduz na criação dinâmica de tarefas ou ativação aperiódica de tarefas criadas estaticamente. Ambos os mecanismos representam uma demanda por recursos que é desconhecida a priori (em tempo de projeto). Não é possível oferecer garantias para uma demanda que não é sequer conhecida.

Estas questões levaram a existência de duas linhas distintas de abordagens. Nas classes de abordagens com garantia (executivo cíclico ou prioridades com teste de escalonabilidade) é oferecida garantia para todas as tarefas, a partir de uma carga limitada e gastando-se todos os recursos que forem necessários. Nos modelos de tarefas usualmente

adotados, todas as tarefas são conhecidas. Tarefas são periódicas ou esporádicas. As tarefas esporádicas, tratadas no pior caso, viram tarefas periódicas. São reservados recursos para a demanda no pior caso de todas as tarefas.

Nas classes de abordagens tipo "melhor esforço" é construído um sistema flexível, que utiliza com eficiência os recursos computacionais disponíveis, porém sem garantia para prazos de execução. Estes sistemas comportam carga dinâmica, onde novas tarefas são criadas ou ativadas a qualquer momento. Os recursos tendem a ser completamente utilizados. Inclusive é possível faltar recursos, ocorrendo então sobrecargas temporárias. Neste caso, não é possível executar todas as tarefas no prazo desejado com a qualidade desejada. Algumas propostas incluem uma garantia probabilista, se a carga for definida em termos também probabilistas.

Reservar recursos para todas as tarefas no pior caso é uma postura muito pessimista. Primeiramente, existe diferença entre o tempo médio e o tempo máximo de execução das tarefas. Muitos algorítmos apresentam curvas de probabilidade para o tempo de execução com uma cauda longa. Ao mesmo tempo, o hardware sempre evoluiu no sentido de prover um menor tempo de execução no caso médio. Na evolução do hardware, nunca houve muita preocupação em diminuir o tempo de execução no pior caso. Exemplos disto são memória virtual, memória cache, processadores com pipeline interna. Memória virtual é um exemplo de mecanismo muito empregado em sistemas convencionais mas raramente utilizado em sistemas de tempo real, por apresentar um péssimo comportamento no pior caso. Esta constatação leva muitos autores a proporem arquiteturas específicas para sistemas de tempo real, sem utilizar as técnicas criadas ao longo do tempo para melhorar o caso médio. No momento em que o projeto considera um cenário pessimista (pior caso), fatalmente ocorre subutilização dos recursos disponíveis no momento da execução (caso médio).

A questão da utilização dos recursos é discutida em [BUR 91]. Burns e Wellings reconhecem que, se a maioria das tarefas tiverem seus deadlines 100% garantidas em tempo de projeto, os recursos do sistema serão fatalmente subutilizados. Isto é uma conseqüência do fato desta garantia ser obtida considerando-se o pior caso para: desempenho do hardware, tempo de processamento das tarefas, tempo de bloqueio entre tarefas. É dito em [BUR 91] que "um desafio para a próxima geração de sistemas é encontrar meios de usar esta capacidade ociosa para melhorar a utilidade ou qualidade dos serviços providos pelo sistema". Esta melhoria poderia representar um aumento da precisão, uma conclusão antecipada das tarefas ou um aumento da confiabilidade. É sugerido que o aumento de precisão é a forma de mais fácil aplicabilidade, tendo computação imprecisa como um possível mecanismo. O ponto chave é como recuperar os recursos reservados mas não utilizados e então aproveitar estes recursos para melhorar a utilidade/qualidade do sistema. Um certo grau de flexibilidade é necessário para aumentar a utilidade do sistema e permitir reconfigurações dinâmicas. Entretanto, esta flexibilidade desejada não elimina a necessidade de garantir deadlines para um conjunto de tarefas.

Muitos textos na bibliografia procuram comparar as diferentes abordagens. Em [KOP 92], Kopetz compara o escalonamento estático com o escalonamento dinâmico. Em [STA 92], Stankovic discute como as diferentes abordagens tratam os recursos além dos processadores. Ele conclui que o maior problema não é o escalonamento do processador, mas sim o tratamento integrado de todos os recursos computacionais do sistema.

Schwan e Zhou [SCH 92] defendem a idéia de um escalonamento dinâmico. Eles argumentam que a maioria dos sistemas tipo tempo real devem estar preparados para lidar com:

- Grandes variações nas taxas de ocorrência dos eventos externos;
- Um número exponencialmente grande de possíveis ocorrências simultâneas de eventos;
- Requisitos quanto à reação do sistema a tais eventos que variam ao longo do tempo;
- Mudanças inesperadas nos recursos computacionais.

É importante destacar que a maioria dos autores admitem que sistemas reais devem empregar abordagens mistas. Por exemplo, sistemas baseados em executivo cíclico podem reservar alguns slots do processador para atender tarefas aperiódicas não garantidas. Sistemas baseados em prioridades e teste de escalonabilidade empregam servidores ou mecanismos como tomada de folga para atender tarefas aperiódicas. Abordagens tipo melhor esforço não descartam a possibilidade de existirem tarefas periódicas garantidas no sistema, cuja escalonabilidade é mantida pelo algorítmo usado em tempo de execução.

Existem fortes argumentos tanto para a necessidade de haver uma previsibilidade determinista em algumas tarefas quanto para a necessidade de alguma flexibilidade na carga do sistema. Esta realidade conduz o estudo dos sistemas tempo real para cenários onde garantia e flexibilidade na carga são, de alguma maneira, compatibilizados.

Esta seção termina com uma tabela que procura resumir as características das diferentes abordagens. Esta tabela foi construída a partir do levantamento bibliográfico descrito neste capítulo. O conteúdo da tabela é, até certo ponto, discutível.. Para comparar especificamente as propostas, seria necessária uma tabela comparativa independente para cada abordagem. Propostas pertencentes à abordagens diferentes trabalham com premissas e objetivos tão distintos, que não podem ser comparadas diretamente.

	Executivo Cíclico	Prioridades	Descarta	Sacrif.ica Deadlines
Modelo de Carga	Menos flexível	Menos flexível	Mais flexível	Mais flexível
Uso de Recursos	Ineficiente	Ineficiente	Eficiente	Eficiente
Previsibilidade	Comportamento conhecido	Prazos conhecidos	Probabilista (carga também)	Probabilista (carga também)

2.6 Conclusões

Neste capítulo foi feita uma revisão de sistemas tempo real e da problemática do escalonamento tempo real. A seção 2.5 procurou sintetizar a discussão existente no que diz respeito às abordagens para este problema e suas caracterísitcas. Uma das abordagens possíveis, a Computação Imprecisa, não foi discutida neste capítulo. Ela será apresentada com detalhes no capítulo 3.

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 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") será suficiente.

As abordagens que foram discutidas neste capítulo são limitadas. 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 algorítmos ainda tendem a ser mais complexos e computacionalmente mais custosos.

3 Computação Imprecisa

3.1 Introdução

Como colocado no capítulo anterior, 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]). Eles 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.

3.2 Aspectos Conceituais

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 abordagens descritas no capítulo 2 que são 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. Este 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.

3.2.1 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 algorítmos 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 algorítmos 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 para a frente ("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 graciosa 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 algorítmos tipo qualquer-tempo ("anytime algorithms"). Um algorítmo tipo qualquer-tempo é formado por refinamentos iterativos que, a qualquer momento, podem ser interrompidos e a melhor resposta até o momento é fornecido. É esperado que a qualidade da resposta melhora na medida em que o algorítmo tiver mais tempo para executar.

3.2.2 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], algorítmos 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 algorítmos 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 91a]. Em [KEN 91] é descrita a linguagem Flex, com suporte específico para este tipo de programação.

3.2.3 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 à 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 3.1.

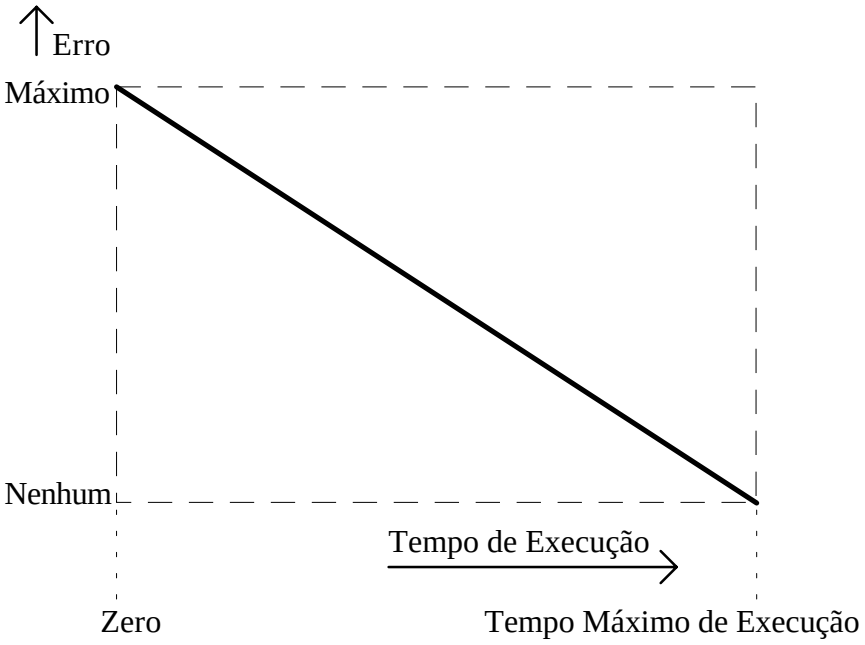

Figura 3.1 - Função de erro na forma de uma reta.

É reconhecido que, na prática, a função de erro não tem sempre o formato de uma reta. Dependendo do algorítmo em questão, esta curva pode ser linear, côncava, convexa ou ainda possuir um formato mais complexo. Por exemplo, um algorítmo para cálculo numérico que realiza iterações que convergem para o resultado possivelmente possui uma curva côncava, como ilustra a figura 3.2. No início da execução do algorítmo, 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.

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 3.3 ilustra a situação onde uma função de melhoramente perde sua capacidade de gerar benefício para o sistema se os dados de entrada forem muito ruins ou muito bons.

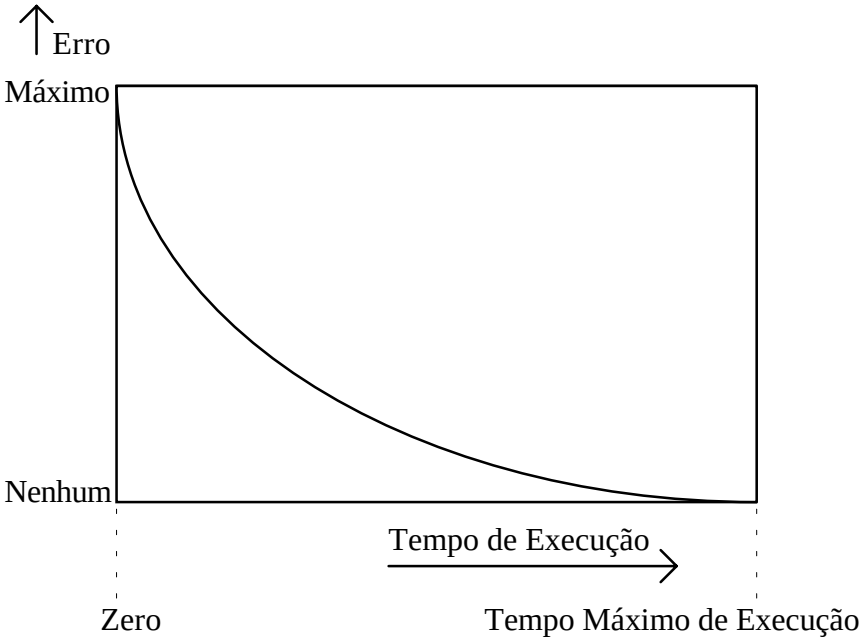

Figura 3.2 - Função de erro na forma de uma curva côncava.

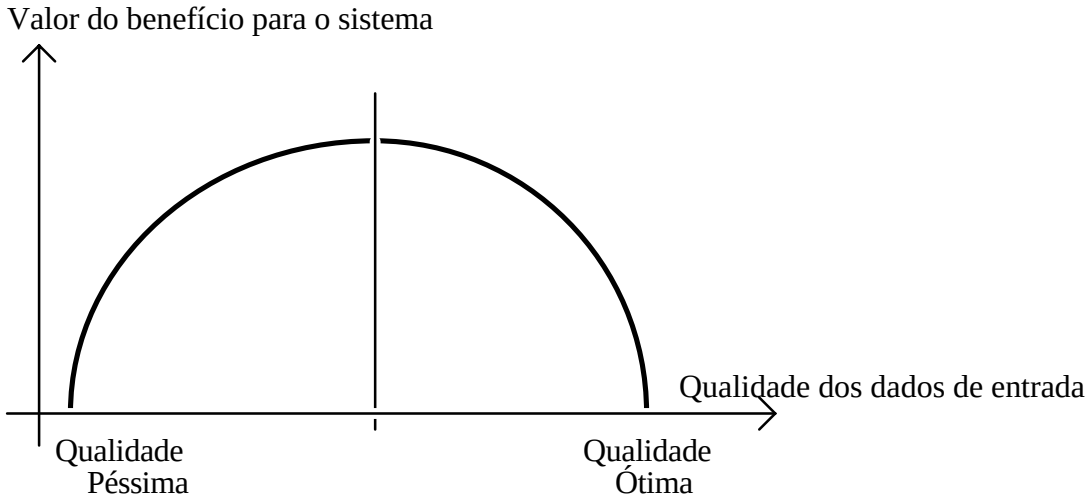

Figura 3.3 - Benefício gerado em função dos dados de entrada.

A figura 3.4 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. Não é de nosso conhecimento nenhum algorítmo na bibliografia que considere a qualidade dos dados de entrada para computar o erro associado com uma parte opcional.

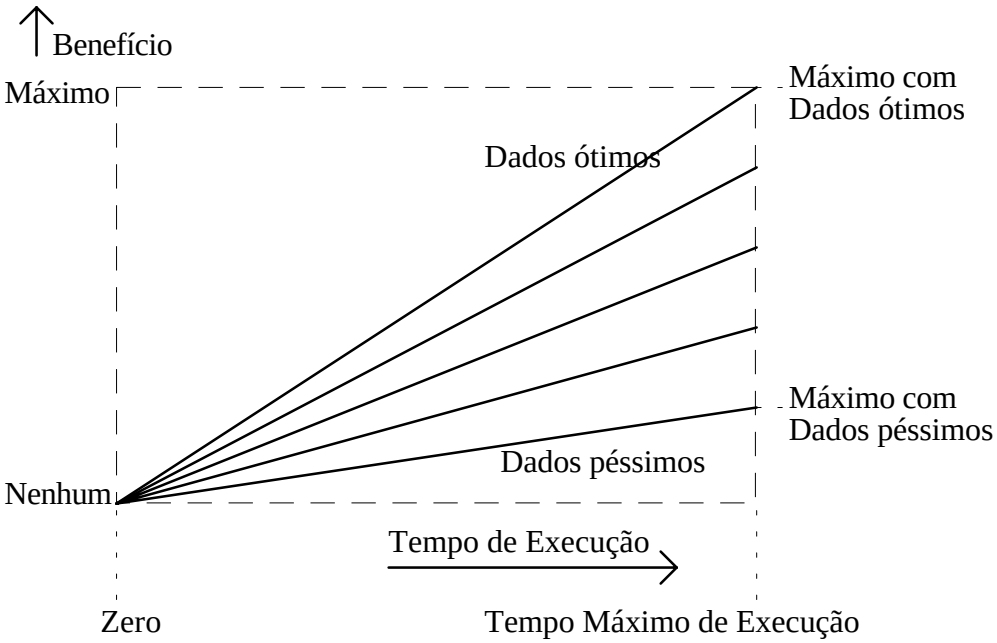

Figura 3.4 - Função de erro na forma de um conjunto de retas.

3.2.4 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.

3.3 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 algorítmos 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.

3.3.1 Considerações Gerais

A análise matemática dos algorítmos de escalonamento requer a formalização dos conceitos de tarefa, parte obrigatória, parte opcional, prazo, benefício, etc. A maioria das propostas modelam cada tarefa ***Ti*** como uma composição de uma parte obrigatória ***Mi*** e uma parte opcional ***Oi***. Existe uma relação de precedência entre ***Mi*** e ***Oi***. Cada tarefa ***Ti*** 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 ***Mi*** e ***Oi*** são os mesmos de ***Ti***. Os tempos de execução de ***Mi*** e ***Oi*** somados resultam no tempo de execução de ***Ti***. Nesta formalização é necessário uma estimativa para o tempo de execução de ***Oi***, mesmo que este seja exageradamente grande. As formas básicos de programação são modeladas da seguinte maneira:

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

3.3.2 Classificação das Propostas

Cada proposta envolvendo Computação Imprecisa é caracterizada pelo:

- Modelo de tarefas considerado;
- Função de erro e seu uso no escalonamento;
- Algorítmos 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 tres tipos de propostas, como ilustra a figura 3.5. 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.

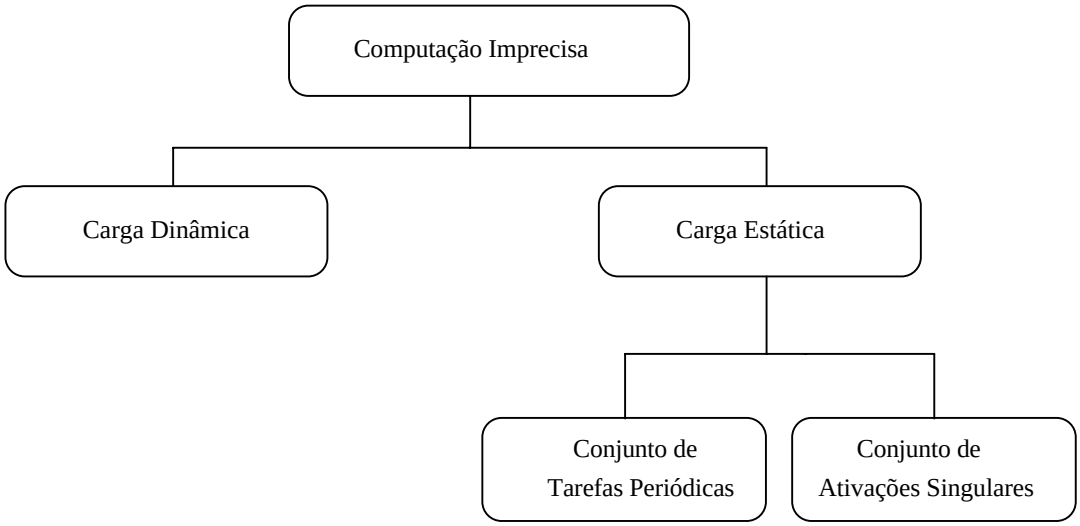

Figura 3.5 - Classificação das Propostas que empregam Computação Imprecisa.

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 algorítmo de escalonamento procura aumentar o benefício gerado pelo sistema da melhor maneira possível. Como esta carga é potencialmente ilimitada, este algorítmo 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 a 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 a 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 3.2.1. 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.

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

- 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 3.2.3 e 3.2.4 trataram da função de erro e seu uso durante o escalonamento. Na verdade, é possível imaginar dezenas de variações com respeito à estes

aspectos. Muitos 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).

3.4 Propostas na Literatura Usando Computação Imprecisa

Nesta seção as propostas presentes na bibliografia são apresentadas. Cada proposta é classificada segundo a taxonomia apresentada na seção anterior e identificada pela referência bibliográfica onde ela pode ser encontrada.

3.4.1 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.

3.4.1.1 Minimiza Erro Total [SHI 92]

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

Cada tarefa ***Ti*** é logicamente decomposta em uma parte obrigatória com tempo de execução ***Mi*** e outra parte opcional com tempo de execução ***Oi***, com ***Mi + Oi = Ci***. A parte opcional somente pode iniciar sua execução após a conclusão da parte obrigatória. Não existem restrições do tipo 0/1.

O erro ***Ei*** no resultado da tarefa ***Ti*** é definido como igual ao tempo de processamento da porção não executada da sua parte opcional. Desta forma, o erro total ***ET*** é o somatório dos erros individuais das tarefas. Em termos matematicos:

$$ET = \sum_{i=1,2,...,N} (Ei)$$

Os algorítmos apresentados no artigo supõe que todas as tarefas possuem o mesmo peso e são independentes. Os algorítmos de escalonamento apresentados são ótimos no

sentido que conseguem concluir as partes obrigatórias sempre antes das deadlines e ainda minimizam o erro total ***ET*** do sistema, a partir das suposições feitas.

As diferenças entre os tres algorítmos tem origem em dois fatores:
- A possibilidade ou não de existirem tarefas off-line, ou seja, que já são conhecidas antes do inicio da execução do sistema;
- A possibilidade ou não de existirem tarefas cujo tempo de pronto ("ready time") é diferente do momento da chegada ("arrival time"), quando o escalonador toma conhecimento da tarefa.

São propostos 3 algorítmos:

- NORA ("No-Off-line tasks and on-line tasks Ready upon Arrival"), para quando não existem tarefas "off-line" e as tarefas "on-line" estão prontas para execução no momento de sua chegada. Apresenta complexidade ***O(N log N)***.

- OAR ("On-line tasks with Arbitrary Ready time"), para quando podem existir tarefas "off-line" e as tarefas "on-line" possuem um tempo de pronto arbitrário, ou seja, algumas tarefas não estão prontas para executar no momento da chegada. Apresenta complexidade ***O(N log^2 N)***.

- ORA ("On-line tasks Ready upon Arrival"), para quando podem existir tarefas "off-line" e as tarefas "on-line" estão prontas para execução na chegada. Apresenta complexidade ***O(N log N)***.

O mecanismo básico dos tres algorítmos é o mesmo. Eles são baseados no EDF ("Earliest Deadline First") e mantém uma lista de reserva para gerenciar a utilização do processador. Primeiro reservam tempo de processador para executar completamente as partes obrigatórias das tarefas conhecidas. Em seguida, o tempo de processador ainda livre é utilizado para executar o máximo possível da parte opcional de cada tarefa. A diferença entre os algorítmos está na estrutura de dados utilizada para manter esta lista de reserva.

3.4.1.2 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.

Novamente cada tarefa ***Ti***, ***i*** entre ***1*** e ***N***, é caracterizada por um momento de liberação ***Ri*** ("release"), um deadline ***Di,*** um tempo de computação para a parte obrigatória ***Mi***, um tempo de computação para parte opcional ***Oi***. O tempo total de execução ***Ci*** é definido como ***Mi+Oi***. Todas as tarefas possuem o mesmo peso. Embora os autores não digam, fica implícito que todas as tarefas estão prontas para execução no momento da liberação.

O erro ***Ei*** no resultado da tarefa ***Ti*** é definido como igual ao tempo de processamento da porção não executada da sua parte opcional. O erro total ***ET*** é o somatório dos erros individuais das tarefas, ponderado pelo peso. Em termos matematicos:

$$ET = \sum_{i=1,2,...,N} (Ei \times Wi)$$

Em [HO 92a] são considerados dois problemas distintos de escalonamento, resultantes de duas definições distintas para o objetivo do escalonamento das partes opcionais: minimizar o erro total ***ET*** e minimizar o número de partes opcionais não executadas.

Quando escalonamento ótimo é definido como aquele que minimiza o erro total ***ET*** do sistema, o problema torna-se NP-hard, como mostrado em [SHI 91]. Em [HO 92a] é feita uma redução deste problema à uma situação para a qual existe solução ótima na bibliografia com complexidade pseudo-polinomial $O(N^5.X^2)$. Aqui, ***N*** é o número de tarefas e ***X*** é o somatório dos tempos de computação das partes opcionais.

Em [HO 92a] é fornecida uma heurística subótima para este mesmo problema, com complexidade computacional $O(N^2)$. As tarefas são ordenadas de forma que o tempo de execução das partes opcionais ***Oi*** fique em uma sequência não crescente. A seguir, é construido um escalonamento a partir da inclusão uma a uma das tarefas. A cada passo, é incluída a tarefa que resulta no menor ***ET***, considerando que as tarefas ainda não incluidas terão suas partes opcionais descartadas. É demonstrado que esta heurística obtém um erro total que é, no máximo, 3 vezes o erro total ótimo.

Também é considerado neste artigo o problema de escalonamento quando o objetivo é minimizar o número de partes opcionais descartadas. Como o modelo de tarefas usado inclui uma restrição do tipo 0/1, o número de partes opcionais descartadas é igual ao número de tarefas não executadas completamente. Em [SHI 91] é mostrado que minimizar descarte é polinomial se todas as partes opcionais tiverem o mesmo tempo de execução. Em [HO 92a] é feita uma redução deste problema à uma situação para a qual existe solução ótima na bibliografia com complexidade $O(N^7)$, para modelos de tarefas onde as partes opcionais possuam tempos de execução diferentes.

Em [HO 92a] é fornecida uma heurística subótima para este problema, com complexidade $O(N^2)$. A heurística empregada aqui é semelhante aquela descrita antes. apenas as tarefas são agora ordenadas de maneira que os tempos de execução das partes opcionais fique não decrescente. É demonstrado que esta heurística obtém um número de partes opcionais descartadas que é, no máximo, 2 vezes o número ótimo.

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

Em [HO 94] são apresentados dois algorítmos 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 algorítmos 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.

O modelo de tarefas é semelhante ao descrito na seção anterior. Cada tarefa ***Ti***, ***i*** entre ***1*** e ***N***, é caracterizada por um momento de liberação ***Ri*** ("release"), uma deadline ***Di,*** um tempo de computação para a parte obrigatória ***Mi***, um tempo de computação para parte opcional ***Oi*** e dois pesos ***Wi*** e ***W'i***. O tempo total de execução ***Ci*** é definido como ***Mi+Oi***. Está implícito no artigo que todas as tarefas estão prontas para execução no momento da liberação. Não existem restrições do tipo 0/1.

Como na proposta anterior, o erro ***Ei*** no resultado da tarefa ***Ti*** é definido como igual ao tempo de processamento da porção não executada da sua parte opcional. O erro total ***ET*** é o somatório dos erros individuais das tarefas, ponderado pelo peso. Ou seja:

$$ET = \sum_{i=1,2,\ldots,N} (Ei \times Wi)$$

Mesmo quando o erro total ***ET*** é minimizado, existe a possibilidade de que a distribuição do erro entre as tarefas seja bastante irregular. Uma forma de atacar este problema é usar como objetivo a minimização do erro máximo normalizado, sujeito à restrição de que ***ET*** foi minimizado. O erro normalizado ***Yi*** associado com a tarefa ***Ti*** é definido como ***Yi = Ei / Oi***. O erro normalizado ***Yi*** pode ser entendido como o erro ***Ei*** ponderado por um peso ***W'i*** igual a ***1 / Oi***. Desta forma, é possível generalizar o objetivo através do emprego de dois pesos: um associado com o erro total (***Wi***) e outro associado com o erro máximo (***W'i***).

O primeiro algorítmo apresentado é ótimo no sentido que ele minimiza o erro máximo das tarefas, ponderados pelos pesos ***W'i***. Nesta situação, os pesos ***Wi*** são ignorados. Sua complexidade computacional é $O(N^2)$ para monoprocessadores e $O(N^3.Log^2\ N)$ para multiprocessadores homogêneos com ***V*** processadores.

O segundo algorítmo é ótimo no sentido que ele minimiza o erro total ***ET*** considerando os pesos ***Wi***, sujeito à restrição de que o erro máximo foi minimizado considerando os pesos ***W'i***. Entre todos os escalonamentos possíveis que possuem um erro ponderado máximo minimizado, este algorítmo encontra um com ***ET*** mínimo. Ele apresenta a mesma complexidade computacional do primeiro algorítmo.

Em [HO 92b] é apresentado um algorítmo que segue o modelo de tarefas descrito. O algorítmo é ótimo no sentido que ele minimiza o erro máximo considerando os pesos ***W'i***, sujeito à restrição de que o erro total ***ET*** foi minimizado considerando os pesos ***Wi***. Sua complexidade computacional é $O(K.N^2)$ para monoprocessadores e $O(K.N^3.Log^2\ N)$ para multiprocessadores homogêneos. Na complexidade, ***K*** representa o número de pesos diferentes existentes.

3.4.2 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.

3.4.2.1 Escalona Conforme Importância [AUD 91c]

Nesta proposta cada tarefa é modelada como uma sequência de 5 fases:

$$\boldsymbol{I + C1 + X + C2 + O},$$

onde ***I*** representa a preparação dos dados de entrada, ***O*** é o envio do resultado ao seu destinatário, ***C1*** e ***C2*** correspondem às computações obrigatórias e ***X*** é uma computação opcional. As fases ***I***, ***C1***, ***C2*** e ***O*** deverão ser analisadas levando em consideração o tempo de execução no pior caso, pois estas devem ser sempre executadas. O mesmo não acontece com a fase ***X***. Esta sequência de 5 fases consegue representar as formas de programação apresentadas na seção 3.2.2. Para efeito de escalonamento, as fases ***I*** e ***C1*** são agrupadas em uma tarefa prólogo e as fases ***C2*** e ***O*** são agrupadas em uma tarefa epílogo. A fase ***X*** forma a parte opcional. Não existem restrições do tipo 0/1.

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 suas deadlines não são garantidas. É suposto no modelo um processador funcionando com escalonador preemptivo.

A folga da tarefa original é divida igualmente entre as tarefas prólogo e epílogo derivadas desta. O tempo de execução do prólogo, somado a sua folga, definem o deslocamento ("offset") do epílogo com relação ao momento de liberação ("release"). Deadline Monotonic é utilizado para definir as prioridades das tarefas ([AUD 90a], [AUD 91a]). A definição de prioridades feita por este algorítmo, neste caso, não é ótima. Isto ocorre porque algumas tarefas (os epílogos) possuem deslocamento ("offset") não nulo. Um teste suficiente e não necessário provido em [AUD 90a] e [AUD 91a] é extendido para o modelo em questão, apresentando complexidade $\boldsymbol{O(N^2)}$ no número de tarefas. Este teste de escalonabilidade leva em consideração os tempos de bloqueio devido à recursos compartilhados e a conseqüente possibilidade de inversão de prioridades.

Na proposta de [AUD 91c], as tarefas opcionais também recebem prioridades fixas, menores que todos os prólogos e epílogos. Estas prioridades são derivadas das respectivas "utilidades para o sistema", uma estimativa do benefício da execução da tarefa para o sistema como um todo. O esquema é bastante simples, tanto na implementação como nos resultados obtidos. Os autores reconhecem que associar um único valor de utilidade para cada parte opcional é muito limitado. Por exemplo, funções monotônicas podem apresentar curvas qualidade do resultado versus tempo não lineares. Isto significa que o benefício gerado pela primeira interação da função monotônica pode melhorar substancialmente a qualidade do resultado, mas as iterações seguintes obterem apenas ganhos marginais.

3.4.2.2 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. Nesta seção, será discutido especificamente o problema de escalonamento das tarefas tipo N-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.

O artigo apresenta uma classe de algorítmos preemptivos, dirigidos por prioridade, que produzem um escalonamento satisfatório com baixo erro médio. O modelo de tarefas adotado considera conjuntos de ***N*** tarefas periódicas imprecisas em um multiprocessador. As tarefas são independentes. Cada tarefa ***Ti***, com ***i*** entre ***1*** e ***N***, é caracterizada por um período ***Pi***, um deadline ***Di***, um tempo total de processamento ***Ci*** e um peso ***Wi***. O deadline ***Di*** é sempre o final do período correspondente à ativação. É suposto que o primeiro período de todas as tarefas inicia no instante zero de tempo, ou seja, não existe deslocamento ("offset") entre as tarefas. O momento de pronto é sempre o início do período correspondente à ativação.

Cada tarefa ***Ti*** é logicamente decomposta em uma parte obrigatória com tempo de execução ***Mi*** e outra parte opcional com tempo de execução ***Oi***. Estes tempos são tais que ***Mi + Oi = Ci***. A parte opcional somente pode iniciar sua execução após a conclusão da parte obrigatória. O erro ***Ei*** no resultado da tarefa ***Ti*** é definido como uma função ***F*** do tempo de processamento da porção não executada da sua parte opcional. Não existem restrições do tipo 0/1.

Uma métrica razoável para o erro associado com uma tarefa em particular é o erro médio ***EM*** desta tarefa. Em [CHU 90], EM é medido ao longo de um intervalo de tempo equivalente ao MMC (mínimo múltiplo comum) dos períodos de todas as tarefas. O erro médio geral ***EMG*** do sistema é definido como:

$$EMG = \sum_{i=1,2,...,N} (Wi \times EMi)$$

O objetivo do escalonamento é encontrar soluções que garantam a conclusão de todas as partes obrigatórias antes do respectivo deadline, ao mesmo tempo que minimiza o ***EMG***.

Em [CHU 90] são utilizadas prioridades preemptivas no escalonamento. As partes obrigatórias possuem prioridades superiores a todas as partes opcionais. Logo, uma parte opcional somente é executada quando não existe nenhuma parte obrigatória pronta para

executar. As partes obrigatórias recebem prioridades conforme o método Rate-Monotonic ([LIU 73]), o que permite verificar em tempo de projeto a sua escalonabilidade.

No modelo de tarefas adotado, o sistema é executado em um multiprocessador. Antes de definir a prioridade de cada tarefa segundo o RM, é feita a alocação das tarefas aos processadores. Depois disto, RM é aplicado a cada processador individualmente.

A alocação das tarefas aos processadores é feita através da técnica conhecida como RM-NF ("rate-monotonic next-fit"). As tarefas são classificadas na ordem crescente de seus períodos. Elas são então alocadas uma a uma, segundo esta ordem, no próximo processador onde a tarefa couber. Uma tarefa cabe em um processador se ela, e as demais tarefas já alocadas à este processador, podem ter suas deadlines garantidas pelo teste de escalonabilidade do RM. Esta alocação considera como tempo de execução da tarefa ***Ti*** o valor ***Mi***. Em outras palavras, a alocação é feita de maneira à garantir apenas a parte obrigatória de cada tarefa.

Durante a execução, cada ativação de uma tarefa inicia com a prioridade definida pelo método descrito antes. No momento que a parte obrigatória foi concluída e vai iniciar a parte opcional, a prioridade da tarefa é reduzida. Para efeitos práticos, tudo acontece como se a parte obrigatória e a parte opcional de cada tarefa possuissem prioridades diferentes. Isto é feito de maneira que todas as partes obrigatórias possuam prioridades superiores à qualquer parte opcional. Desta forma, as partes obrigatórias são sempre concluídas antes do deadline, independentemente do escalonamento das partes opcionais.

Heurísticas são empregadas para definir as prioridades das partes opcionais. Diversas opções são consideradas, embora nenhuma seja ótima no sentido de que minimiza o ***EMG***. Inclusive, estas heurísitcas podem levar a escalonamentos com ***EMG*** não zero, mesmo quando é possível obter um escalonamento preciso aplicando RM sobre as tarefas como um todo (usando ***Ci*** no teste de escalonabilidade). Quando isto acontece, RM clássico deve ser utilizado. Em [CHU 90] são consideradas as seguintes heurísticas:

- "Least-Utilization" associa prioridades fixas maiores às partes opcionais com menor fator de utilização (tempo de computação dividido pelo período), ponderado pelo peso. Esta heurística apresenta melhores resultados quando a função de erro ***F*** é linear.

- "Least-Attained-Time" associa prioridades variáveis maiores às partes opcionais que receberam menor tempo de processador até o momento. Esta heurísitca apresenta melhores resultados quando a função de erro ***F*** é convexa.

- "First-Come-First-Serve" executa cada parte opcional até o fim, pela ordem de chegada. Apresenta melhores resultados quando a função de erro ***F*** é côncava.

- "Shortest-Period" associa prioridades fixas maiores às partes opcionais com período mais curto. Heurística apropriada para quando o formato da função erro ***F*** não é conhecido.

- "Earliest-Deadline" associa prioridades variáveis maiores às partes opcionais com a deadline mais próxima. Heurística apropriada para quando o formato da função erro ***F*** não é conhecido.

Em [CHU 90] são apresentados gráficos com o resultado de uma análise feita a partir de uma família de funções de erro ***F***. Fica claro que o formato da função de erro ***F*** é crucial na escolha da heurística a ser empregada.

3.4.2.3 Minimiza Erro Acumulado Máximo [CHU 90]

Ainda considerando as propostas em [CHU 90], nesta seção será discutido especificamente o problema de escalonamento das tarefas tipo AC.

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.

O modelo de tarefas adotado considera conjuntos de ***N*** tarefas periódicas imprecisas em um monoprocessador. As tarefas são independentes. Cada tarefa ***Ti***, com ***i*** entre ***1*** e ***N***, é caracterizada por um período ***Pi***, uma deadline ***Di***, um tempo total de processamento ***Ci*** e um peso ***Wi***. É suposto que o primeiro período de todas as tarefas inicia no instante zero de tempo, ou seja, não existe deslocamento ("offset") entre as tarefas. O momento de pronto é sempre o início do período correspondente à ativação. A deadline ***Di*** é sempre o final do período correspondente à ativação.

Cada tarefa ***Ti*** é logicamente decomposta em uma parte obrigatória com tempo de execução ***Mi*** e outra parte opcional com tempo de execução ***Oi***. Estes tempos são tais que ***Mi + Oi = Ci***. A parte opcional somente pode iniciar sua execução após a conclusão da parte obrigatória. O erro ***Ei*** no resultado da tarefa ***Ti*** é definido como uma função do tempo de processamento da porção não executada da sua parte opcional. Em outras palavras, o erro de uma tarefa é igual ao tempo de processamento que faltou para concluir a sua parte opcional.

Cada tarefa ***Ti*** possui, a cada momento, um erro acumulado ***EAi***. O erro acumulado ***EAi*** é definido como o somatório de ***Ei*** desde a última vez que a tarefa executou completamente (zerou o erro acumulado). Uma falha temporal ocorre no sistema sempre que ***EAi*** ultrapassa um valor limite ***Qi***, parte da especificação do sistema. Logo, uma solução para o escalonamento deste tipo de tarefa deve garantir que:

- As partes obrigatórias são sempre concluídas antes do deadline;
- Para cada tarefa ***Ti***, uma ativação da parte opcional é completamente executada antes que ***EAi*** ultrapasse o valor máximo especificado.

Em [CHU 90] é apresentado um algorítmo de escalonamento que considera uma função de erro tal que:

Ei = 0 quando a parte opcional é concluída;
Ei = 1 quando a parte opcional não é concluída.

Nestas condições, requerer que o erro acumulado pela tarefa ***Ti*** permaneça abaixo de determinado valor ***Qi*** é o mesmo que requerer que ao menos uma ativação entre quaisquer ***Qi*** ativações de ***Ti*** seja executada completamente, para ***i*** entre ***1*** e ***N***. O objetivo de um algorítmo de escalonamento nesta situação é garantir os prazos das partes obrigatórias e providenciar para que cada parte opcional consiga executar completamente antes de passarem ***Qi*** períodos (ativações da tarefa ***Ti***). Quando uma parte opcional executa completamente, ela zera o erro ***EAi*** que havia acumulado até então.

O algorítmo apresentado reduz ainda mais o problema, considerando tarefas com mesmo período e mesmo limite para o erro acumulado. Ou seja, para qualquer ***i***, todos os ***Ti*** são iguais e para qualquer ***i***, todos os ***Qi*** são iguais. É mostrado que encontrar um escalonamento satisfatório neste caso é um problema NP-completo. O artigo sugere então uma heurística com resultados subótimos.

A heurística é baseada no algorítmo tamanho monotônico ("length-monotone"). No início de cada período do conjunto de tarefas (todas possuem o mesmo período) é reservado tempo de processador para as partes obrigatórias. O tempo de processador restante é reservado para as partes opcionais. As partes opcionais são atendidas conforme a ordem decrescente de seus tempos de execução, ao longo de um intervalo de tempo suficientemente grande para executar cada uma exatamente uma vez. Os autores desenvolvem um teste de escalonabilidade para este algorítmo, a partir das premissas assumidas.

3.4.2.4 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 a existência de uma parte opcional em cada tarefa. Localmente é empregado EDF ("Earliest Deadline First") ou RM ("Rate-Monotonic") para escalonar cada processador.

O modelo de tarefas considerado define uma carga composta por ***N*** tarefas periódicas independentes, conhecidas em tempo de projeto. Cada tarefa é decomposta em uma parte obrigatória e uma parte opcional. Não existem restrições do tipo 0/1. As tarefas são executadas em um multiprocessador composto por ***V*** processadores idênticos. É suposto que a interconexão entre os processadores é tal que suporta difusão confiável.

É suposto que o sistema a ser escalonado possui requisitos de tolerância à faltas. O objetivo desta proposta é tolerar faltas no hardware, supondo que o software é livre de erros. Quando um processador falha, ocorre uma redução dos recursos disponíveis. O sistema reage com uma mudança no modo de operação. O resultado desta mudança é uma redução no tempo de execução das partes opcionais e uma redução no número de réplicas das tarefas. Desta forma, o sistema adapta-se a uma situação onde menos recursos estão

disponíveis. Neste texto, serão ignorados os detalhes do mecanismo de tolerância à faltas, sendo discutidos apenas os aspectos ligados à Computação Imprecisa.

São definidos ***V*** modos de operação para o sistema, considerando que existe a possibilidade de haverem entre ***1*** e ***V*** processadores operacionais a cada momento. Para cada modo de operação, é empregada uma alocação diferente. Logo, do ponto de vista da alocação e escalonamento tempo real, são na verdade ***V*** problemas distintos.

A proposta incorpora Computação Imprecisa à técnica de replicação e mascaramento. Existem ***N*** tarefas imprecisas periódicas, cada uma podendo ou não ser replicada. Para cada modo de operação (definido pelo número de processadores operacionais), a especificação do sistema determina o número desejado de réplicas para cada tarefa. As réplicas de uma mesma tarefa devem ser alocadas em processadores distintos.

Cada tarefa ***Ti*** é caracterizada pelo período ***Pi***, tempo total de execução ***Ci***, peso ***Wi*** indicando a importância relativa da tarefa e pelo vetor ***Ri*** com ***V*** elementos. Cada elemento ***Ri[j]***, com ***j*** entre ***1*** e ***V***, representa o número de réplicas desejado para a tarefa ***Ti*** quando existem ***j*** processadores operacionais. Cada tarefa ***Ti*** é decomposta em parte obrigatória com tempo de execução ***Mi*** e parte opcional com tempo de execução ***Oi***, tal que ***Mi + Oi = Ci***. A parte opcional somente pode iniciar após a conclusão da parte obrigatória. Toda liberação ("release") de tarefa (inclusive a primeira) ocorre obrigatoriamente no início de um período. O deadline da tarefa é sempre igual ao final do respectivo período. Não existem restrições do tipo 0/1.

O objetivo do algorítmo proposto no artigo é alocar as tarefas aos processadores de forma que todos os deadlines sejam cumpridas. Isto deve ser feito considerando que não existe migração de tarefas e duas réplicas da mesma tarefa não podem ser alocadas ao mesmo processador.

Localmente, as tarefas alocadas a um mesmo processador serão escalonadas segundo a abordagem baseada em prioridades. O artigo sugere a utilização de EDF ou RM para atribuir prioridades às tarefas. Em tempo de projeto, é aplicado o teste de escalonabilidade do método empregado, que garante o cumprimento das deadlines de todas as tarefas. São empregados os testes de escalonabilidade associados com a utilização do processador, descritos em [LIU 73].

A alocação e o teste de escalonabilidade devem ser feitos considerando como tempo efetivo de execução de cada réplica da tarefa ***Ti*** o valor ***Xi***. Este valor ***Xi*** deve estar entre o valor ***Mi*** (tempo de execução apenas da parte obrigatória) e o valor ***Ci*** (tempo de execução da tarefa completa). Uma vez que cada tarefa possui uma parte opcional, o seu tempo de execução é negociável. É responsabilidade do algorítmo de alocação descrito definir quanto da parte opcional de cada tarefa será executado (o quando ***Xi*** se aproxima de ***Ci***). Isto é feito ao mesmo tempo em que é definida a alocação das réplicas aos processadores. Cada réplica de ***Ti*** recebe o mesmo tempo ***Xi*** de processador, ou seja, todas as réplicas de uma dada tarefa fornecem um resultado com a mesma qualidade.

Dois conceitos são básicos para o algorítmo de alocação apresentado. A utilização ***Ui*** associada com a tarefa ***Ti*** é definida como ***Ui = Xi / Pi***. O benefício global do sistema,

chamado ***TWU*** ("total weighted utilization") é definido como o somatório das utilizações das tarefas, ponderado pelo peso individual de cada tarefa. Em termos matemáticos:

$$TWU = \sum_{i=1,2,...,N} (Wi \times Ui)$$

Na medida que o algorítmo tem liberdade para escolher ***Xi***, ***i*** entre ***1*** e ***N***, existem muitas soluções possíveis. É considerada uma solução ótima aquela que, além de garantir todas as deadlines e respeitar as restrições, maximiza o valor ***TWU***.

O algorítmo de alocação possui 3 fases. Na primeira fase, um algorítmo ganancioso ("greedy") computa um valor máximo para o ***TWU*** e um valor máximo para a utilização de cada tarefa. Estes valores máximos são obtidos através do relaxamento de alguns requisitos do problema. Na segunda fase, dois algorítmos baseados em heurísticas geram duas possíveis alocações para o conjunto de tarefas. Eles empregam o valor máximo para a utilização de cada tarefa. A alocação mais próxima da alocação ótima é selecionada. Como o valor máximo de utilização de cada tarefa foi utilizado, é possível que alguns processadores estejam com sobrecarga, ou seja, o teste de escalonabilidade rejeite o conjunto de tarefas alocado. Na terceira fase, a utilização de cada tarefa é ajustada de maneira que os conjuntos de tarefas alocados a cada processador sejam escalonáveis. Isto é feito procurando maximizar o ***TWU***.

Os problemas são formulados como programas lineares e resolvidos através de algorítmos eficientes conhecidos da programação linear. A complexidade computacional resultante é $O(N^3)$. O artigo ainda apresenta uma avaliação do algorítmo através de análise estocástica e simulação de Monte Carlo.

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.

3.4.3 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 completitude do levantamento.

3.4.3.1 Minimiza Erro Total [LIU 91]

Em [LIU 91] são apresentados dois algorítmo para escalonar conjuntos de ***N*** tarefas imprecisas. Cada tarefa ***Ti***, com ***i*** entre ***1*** e ***N***, é caracterizada por um momento de pronto ***Ri*** ("ready time"), uma deadline ***Di***, um tempo total de processamento ***Ci*** e um peso ***Wi***. O modelo de tarefas adotado permite relações de precedência entre as tarefas do conjunto.

O modelo de tarefas básico é o mesmo para os dois algorítmos. Cada tarefa ***Ti*** é logicamente decomposta em uma parte obrigatória com tempo de execução ***Mi*** e outra

parte opcional com tempo de execução ***Oi***. Estes tempos são tais que ***Mi + Oi = Ci***. A parte opcional somente pode iniciar sua execução após a conclusão da parte obrigatória. O conjunto é considerado satisfatoriamente escalonável ("feasibly scheduled") se todas as partes obrigatórias puderem ser concluídas antes dos respectivos deadlines.

O erro ***Ei*** no resultado da tarefa ***Ti*** é definido como igual ao tempo de processamento da porção não executada da sua parte opcional. Em outras palavras, o erro de uma tarefa é igual ao tempo de processamento que faltou para concluir a sua parte opcional. Não existem restrições do tipo 0/1. Desta forma, o erro total ***ET*** é o somatório dos erros individuais das tarefas, cada um multiplicado pelo peso (importância) da respectiva tarefa. Em termos matematicos:

$$ET = \sum_{i=1,2,\ldots,N} (Wi \times Ei)$$

Os algorítmos propostos neste artigo empregam uma variação do EDF, com complexidade ***N.log(N)***. Nesta variação, toda tarefa que não tiver sido concluída no momento do deadline é imediatamente terminada (abortada). Como o algorítmo é baseado no EDF, na escala de execução resultante pode haver a preempção de tarefas. Para determinar se um conjunto de tarefas é satisfatoriamente escalonável, basta empregar esta variação do EDF na construção de uma escala de execução considerando apenas as partes obrigatórias, ignorando as partes opcionais. Se nehuma deadline for perdida, o conjunto é satisfatoriamente escalonável.

O primeiro algorítmo trata da situação onde todas as tarefas possuem o mesmo peso e são executadas em um monoprocessador. Ele é ótimo no sentido que consegue escalonar todas as partes obrigatórias e ainda obter o menor erro total ***ET*** possível. Sua complexidade é ***O(N log N)***, idêntica ao EDF.

Este algorítmo é composto por 3 fases. Na primeira fase, ele é executada considerando todas as tarefas como se fossem completamente opcionais. Ele constrói uma escala de execução segundo a variação de EDF citada antes. Se todas as deadlines forem cumpriodas, o conjunto é escalonável com erro zero. Caso contrário, passa para a segunda fase. Na segunda fase, constrói uma escala de execução segundo a mesma variação do EDF, considerando apenas as partes obrigatórias das tarefas. Se existir pelo menos uma tarefa que perdeu a deadline, não é possível sequer escalonar as partes obrigatórias. Na terceira fase, a escala de execução criada na segunda fase é transformada de maneira à incluir partes opcionais com o objetivo de minimizar o erro total do conjunto.

Este algorítmo pode também ser modificado para escalonar tarefas independentes de pesos iguais em multiprocessador composto por ***V*** processadores idênticos. A complexidade passa então para ***O(V.N + N.Log N)***.

O segundo algorítmo é utilizado quando as tarefas possuem pesos diferentes e executam em monoprocessador. Ele é baseado no LWF ("Largest-Weight First"), que procura escalonar antes as tarefas com maior peso. Este algorítmo também é ótimo. Embora os autores afirmem que o algorítmo apresenta complexidade ***O(N^2)***, em [HO 92a] é colocado que sua complexidade computacional é na verdade ***O(N^2.Log.N)***.

3.4.3.2 Minimiza Erro Total [SHI 89]

Em [SHI 89] é apresentado um algorítmo para escalonar conjuntos de N tarefas imprecisas em um monoprocessador. O modelo de tarefas é o mesmo da proposta apresentada em [LIU 91], descrita na seção anterior.

O algorítmo apresentado é ótimo no sentido que ele sempre encontra uma escala de execução que atende os requisitos das tarefas, se tal escala existir, ao mesmo tempo em que minimiza o erro total ET das tarefas do conjunto. A abordagem empregada é baseada na teoria de grafos. O problema de escalonamento é transformado em um problema de fluxo em grafo. Sua complexidade é $O(N^6)$ caso as tarefas possuam pesos diferentes. Para o caso particular onde todas as tarefas possuem peso igual a 1, a complexidade computacional fica $O(N^2 log^2 N)$.

O artigo mostra ainda que, caso as tarefas sejam independentes (sem relações de precedência), este mesmo algorítmo pode ser adaptado para um multiprocessador com V processadores idênticos. O algorítmo continua ótimo e com a mesma complexidade computacional.

3.5 Comparação entre as Propostas

As propostas apresentadas na seção anterior não esgotam a bibliografia existente, mas ilustram os resultados mais significativos alcançados na área. As tabelas abaixo comparam as propostas listadas no texto.

Tabela 1 - Carga Estática, Conjuntos de Ativações Singulares

	[SHI 89]	[SHI 89]	[SHI 89]	[SHI 89]	[LIU 91]	[LIU 91]	[LIU 91]
Complexidade	$N^2.Log^2N$	N^6	$N^2.Log^2N$	N^6	$N.Log\ N$	$V.N+N.Log\ N$	$N^2.Log\ N$
Objetivo							
Importância							
Erro total	X	X	X	X	X	X	X
Descarte							
Erro médio (periód.)							
Erro max. acumulado							
Erro máximo							
Erro máximo e total							
Erro total e máximo							
Carga							
Ativações singulares	X	X	X	X	X	X	X
Ativações dinâmicas							
Tarefas periódicas							
Paralelismo							
Monoprocessador	X	X			X		X
Multiprocessador			X	X		X	
Peso das tarefas							
Iguais	X		X		X	X	
Diferentes		X		X			X
Restrição tipo 0/1							
Sim							
Não	X	X	X	X	X	X	X
Relações entre tarefas							
Precedência	X	X			X		X
Recursos							
Tipo de algorítmo							
Preemptivo	X	X	X	X	X	X	X
Não preemptivo							
Atende ao objetivo							
Ótimo	X	X	X	X	X	X	X
Subótimo							
Momento da execução do algorítmo							
Tempo de projeto	X	X	X	X	X	X	X
Tempo de execução							

Tabela 2 - Carga Dinâmica, Tarefas com Restrição Tipo 0/1

	[HO 92a]	[HO 92a]	[HO 92a]
Complexidade	N^2	N^2	N^7
Objetivo			
Importância			
Erro total	X		
Descarte		X	X
Erro médio (periód.)			
Erro max. acumulado			
Erro máximo			
Erro máximo e total			
Erro total e máximo			
Carga			
Ativações singulares			
Ativações dinâmicas	X	X	X
Tarefas periódicas			
Paralelismo			
Monoprocessador	X	X	X
Multiprocessador			
Peso das tarefas			
Iguais	X	X	X
Diferentes			
Restrição tipo 0/1			
Sim	X	X	X
Não			
Relações entre tarefas			
Precedência			
Recursos			
Tipo de algorítmo			
Preemptivo	X	X	X
Não preemptivo			
Atende ao objetivo			
Ótimo			X
Subótimo	X	X	
Momento da execução do algorítmo			
Tempo de projeto			
Tempo de execução	X	X	X

Tabela 3 - Carga Dinâmica, Tarefas sem Restrição Tipo 0/1

	[SHI 92]	[HO 92b]	[HO 92b]	[HO 94]	[HO 94]	[HO 94]	[HO 94]
Complexidade	$N.Log\ N$	$K.N^2$	$K.N^3.Log^2\ N$	N^2	$N^3.Log^2\ N$	N^2	$N^3.Log^2\ N$
Objetivo							
Importância							
Erro total	X						
Descarte							
Erro médio (periód.)							
Erro max. acumulado							
Erro máximo				X	X		
Erro máximo e total						X	X
Erro total e máximo		X	X				
Carga							
Ativações singulares							
Ativações dinâmicas	X	X	X	X	X	X	X
Tarefas periódicas							
Paralelismo							
Monoprocessador	X	X		X		X	
Multiprocessador			X		X		X
Peso das tarefas							
Iguais	X						
Diferentes		X	X	X	X	X	X
Restrição tipo 0/1							
Sim							
Não	X	X	X	X	X	X	X
Relações entre tarefas							
Precedência							
Recursos							
Tipo de algorítmo							
Preemptivo	X	X	X	X	X	X	X
Não preemptivo							
Atende ao objetivo							
Ótimo	X	X	X	X	X	X	X
Subótimo							
Momento da execução do algorítmo							
Tempo de projeto							
Tempo de execução	X	X	X	X	X	X	X

Tabela 4 - Carga Estática, Conjuntos de Tarefas Periódicas

	[YU 92]	[CHU 90]	[CHU 90]	[AUD 91c]
Complexidade	N^3			N^2
Objetivo				
Importância				X
Erro total	X			
Descarte				
Erro médio (periód.)		X		
Erro max. acumulado			X	
Erro máximo				
Erro máximo e total				
Erro total e máximo				
Carga				
Ativações singulares				
Ativações dinâmicas				
Tarefas periódicas	X	X	X	X
Paralelismo				
Monoprocessador			X	X
Multiprocessador	X	X		
Peso das tarefas				
Iguais				
Diferentes	X	X	X	X
Restrição tipo 0/1				
Sim			X	
Não	X	X		X
Relações entre tarefas				
Precedência				X
Recursos				X
Tipo de algorítmo				
Preemptivo	X	X	X	X
Não preemptivo				
Atende ao objetivo				
Ótimo				
Subótimo	X	X	X	X
Momento da execução do algorítmo				
Tempo de projeto	X	X	X	X
Tempo de execução			X	

Através destas tabelas comparativas, é possível afirmar que as propostas encontradas na bibliografia utilizão, em geral, modelos de tarefas bastante simples. Em particular, recursos além do pocessador não são normalmente considerados. Além disto, muitas propostas não suportam relações de precedência. O erro é geralmente modelado como um valor proporcional ao tempo de computação previsto para a parte opcional que não foi executada. Também é possível afirmar que as propostas encontradas na literatura cobrem apenas uma pequena parcela do espectro de possíveis usos para as funções de erro das tarefas no escalonamento.

Não existem na bibliografia muitos trabalhos analisando o emprego de distribuição e múltiplas versões para implementar Computação Imprecisa. Em [KOP 89] aparece um dos poucos empregos de Computação Imprecisa no contexto distribuído. 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. Cada versão executa em um processador diferente. A cada execução, 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.

3.6 Conclusões

O capítulo 2 descreveu o dilema existente na área de tempo real, entre ter garantia para o prazo das tarefas ou ter adaptabilidade e eficiência no uso dos recursos. Neste capítulo foi descrita a técnica Computação Imprecisa, que 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. Algumas formas de programação também foram propostas, porém de forma isolada, desvinculadas de métodos para a especificação e o projeto dos sistemas. Um dos poucos trabalhos existentes neste sentido é o ambiente de desenvolvimento PERTS ([LIU 93]). Também não existem trabalhos na literatura discutindo, sob a ótica da engenharia de software, a utilidade ou aplicabilidade das funções de erro e seu uso no escalonamento das partes opcionais. Em [AUD 91c] é colocado que "o uso disseminado de técnicas, tais como Computação Imprecisa, somente acontecerá se elas forem integradas aos métodos usuais de engenharia de software".

4 A Proposta do Trabalho: O Emprego da Computação Imprecisa em Sistemas de Tempo Real Distribuídos

4.1 Introdução

Este capítulo contém a proposta para o trabalho a ser desenvolvido como tese de doutorado. Inicialmente, são feitas algumas considerações que servem de base para justificar o tema escolhido. Logo em seguida, são apresentadas a proposta da tese em si e a relevância do tema proposto. No final do capítulo é descrito o plano de trabalho para o restante do curso, incluindo: metodologia a ser seguida, produtos finais esperados, recursos necessários, próximas etapas e cronograma.

4.2 Motivações Para a Proposta

Como visto nos capítulos 2 e 3, o principal dilema das abordagens existentes para a construção de sistemas de tempo real é a escolha entre "garantia" e "adaptabilidade". Computação Imprecisa oferece uma alternativa, na medida que é uma abordagem mista. Entretanto, existem diversas limitações nas propostas existentes dentro da abordagem Computação Imprecisa. Isto é particularmente verdade quando é considerado o seu emprego no contexto de sistemas distribuídos. São questões que permanecem em aberto e serão esboçadas nesta seção.

Alocação de tarefas imprecisas em ambiente distribuído

Existem muito poucas publicações na literatura que tratam da integração entre Sistemas Distribuídos e Computação Imprecisa. Esta integração pode ocorrer de diferentes formas. Uma delas é empregar tarefas imprecisas tradicionais em um ambiente distribuído.

No contexto de sistemas distribuídos, o escalonamento tempo real é resolvido em dois níveis: a alocação das tarefas e o escalonamento local dos processadores. Em algumas aplicações, o problema de alocação é ainda complicado pela exigência de alocação simultânea ou excludente de tarefas no mesmo processador, como descrito no capítulo 2. Como pode ser observado no capítulo 3, a maioria dos algorítmos para escalonamento de tarefas imprecisas não considera o problema de alocação de tarefas imprecisas em ambiente distribuído. Fica claro que os modelos de tarefas suportados pelos algorítmos de escalonamento existentes são limitados. Eles precisam ser ampliados para lidar com o problema de alocação em sistemas distribuídos.

Aproveitamento da distribuição na própria técnica

Uma outra maneira de integrar Computação Imprecisa e Sistemas Distribuídos é aproveitar a distribuição no próprio mecanismo da Computação Imprecisa. Com raras exceções, toda a bibliografia implementa Computação Imprecisa a partir de tarefas imprecisas. A opção "qualidade versus recursos usados" é considerada apenas localmente. Em um ambiente distribuído, seria natural extender o conceito de tarefa imprecisa para o conceito de grupo de tarefas impreciso. Neste caso, a opção "qualidade versus recursos usados" passa a ser feita considerando-se o resultado de um grupo de tarefas e não de uma tarefa individualmente. Por exemplo, cada tarefa do grupo poderia implementar uma versão

diferente do mesmo algorítmo (replicação funcional), com diferentes curvas "qualidade do resultado versus tempo de execução". É razoável esperar dificuldades adicionais na implementação de versões distribuídas para os algorítmos que hoje existem para um contexto local. Não existe na bibliografia uma análise das vantagens e desvantagens de implementar Computação Imprecisa a partir de grupos de tarefas imprecisos.

Uma dificuldade inerente aos sistemas distribuídos é o custo da comunicação entre diferentes computadores. É necessário determinar o que este custo representa face aos algorítmos de escalonamento adotados. Muitos esquemas serão provavelmente inviabilizados em função do custo da comunicação.

Compartilhamento de recursos

Como pode ser observado nas tabelas do capítulo 3, a maioria dos algorítmos para escalonamento de tarefas imprecisas não considera o compartilhamento de recursos além dos processadores. Entretanto, em aplicações reais é natural esperar que tarefas compartilhem recursos. O impacto deste compartilhamento sobre os algorítmos de escalonamento empregados não pode ser ignorado.

Relações de precedência entre tarefas

Muitos algorítmos de escalonamento apresentados no capítulo 3 também não suportam relações de precedência entre tarefas. Relações de precedência permitem o emprego do conceito de atividade. Atividades foram definidas no capítulo 2 como conjunto de tarefas com relações de precedência entre si, representadas por grafos acíclicos. O fato da aplicação estar organizada em termos de atividades e não de tarefas independentes altera tanto o problema de alocação quanto o problema de escalonamento local. Fica claro que os modelos de tarefas suportados pelos algorítmos de escalonamento existentes são limitados e precisam ser ampliados também neste sentido.

Deadline associada com atividade e não com tarefa

Quando as tarefas da aplicação são independentes, é natural associar um deadline com cada tarefa individualmente. No momento que existem relações de precedência, passa a ser mais realista associar deadline com atividades e não mais com tarefas individuais. Isto decorre do fato das deadlines estarem associadas com reações à estimulos externos. Estas reações acontecem normalmente como o resultado do processamento de uma atividade inteira, e não como o resultado de uma tarefa individual.

Quando o deadline está associado com uma atividade e não com cada tarefa individual, o problema da Computação Imprecisa se modifica. Ele passa a ser "sacrificar o tempo de execução de algumas tarefas visando manter o deadline da atividade como um todo". Existe aqui a possibilidade de hierarquizar o conceito de erro/benefício. Cada tarefa contribui com um erro/benefício para a atividade. Por sua vez, a atividade contribui com um erro/benefício para o sistema como um todo. O conceito de erro/benefício hierarquizado pode ser empregado na construção de algorítmos de escalonamento.

Função de erro

Um dos principais problemas na Computação Imprecisa é como representar o erro associado com a não execução de uma parte opcional. Esta representação é usada nas sobrecargas para decidir quais partes opcionais serão sacrificadas. A grande maioria dos algorítmos propostos na bibliografia considera apenas este erro como linearmente proporcional ao tempo de execução que faltou para concluir a parte opcional. Alguns poucos trabalhos usam curvas côncavas ou convexas. A rigor, é possível imaginar uma proposta onde cada tarefa é executada um número de vezes e, a partir destas execuções, obtida uma curva "qualidade do resultado versus tempo de execução" ("profile") para a sua parte opcional. Embora o erro como função linear seja confortável para uma análise matemática algébrica, suas limitações são evidentes. Muitos algorítmos, tais como os utilizados para cálculo numérico baseado em aproximações sucessivas, melhoram rapidamente a qualidade do resultado no início da execução. Porém, eles fazem apenas pequenos refinamentos no final. É um caso onde uma curva côncava seria mais apropriada.

Também foi citado no capítulo 3 a questão da qualidade dos dados de entrada. Obviamente, o benefício que uma tarefa é capaz de trazer ao sistema depende da qualidade dos seus dados de entrada. Este aspecto é ignorado nas propostas existentes na bibliografia, especialmente nos algorítmos de escalonamento.

Uso da função de erro no escalonamento

Os modelos tradicionais de Computação Imprecisa permitem ao usuário especificar o que é obrigatório e o que é opcional no sistema. As partes obrigatórias são garantidas, em tempo de projeto, através de um teste de escalonabilidade. Desta forma, o usuário tem a certeza de que o sistema exibirá um comportamento, no mínimo, com a qualidade obrigatória especificada.

Em tempo de execução, os mecanismos da Computação Imprecisa são utilizados para selecionar quais partes opcionais serão sacrificadas. Isto acontece em situações de sobrecarga, quando não existem recursos suficientes para executar todas as tarefas completamente. Nestas situações, haverá uma variação na qualidade observada do sistema, como um todo, em função do sacrifício de algumas partes opcionais. Esta variação é visível ao usuário. Logo, a forma como a qualidade do sistema varia na sobrecarga deveria ser determinada pela especificação do sistema. E desta especificação deveriam ser retirados os critérios ou objetivos a serem utilizados pelo escalonador, em tempo de execução, com respeito as partes opcionais.

Nos modelos tradicionais de Computação Imprecisa, existem mecanismos que permitem algum controle do projetista sobre o comportamento do sistema em sobrecarga. Especificamente, tarefas possuem pesos e existe uma função de erro associada com a não execução completa de uma parte opcional. As partes opcionais são geralmente escalonadas com o objetivo de minimizar o somatório dos erros das tarefas individuais.

Entretanto, não existe na literatura uma discussão sobre os méritos destes mecanismos. Em particular, não está claro se o conceito de erro total do sistema, como um somatório dos erros das tarefas individuais, é capaz de implementar uma política adequada para o tratamento da sobrecarga. O mecanismo usado no escalonamento das partes

opcionais deve refletir o fato do comportamento observável do sistema ser determinado por conjuntos de tarefas, dependentes entre si.

Por exemplo, imagine um sistema onde uma característica opcional ***Si***, visível ao usuário, é fornecida pelas partes opcionais das tarefas ***Ti*** e ***Tj***, quando executadas completamente. Suponha agora que, em uma situação de sobrecarga, a parte opcional da tarefa ***Ti*** não foi executada. Suponha ainda que a parte opcional da tarefa ***Tj*** sozinha não é capaz de fornecer a característica ***Si***. Neste caso, não existe mais sentido em executar a parte opcional da tarefa ***Tj***, até que a parte opcional da tarefa ***Ti*** possa também ser executada. O sistema variou para um nível onde a característica opcional ***Si*** não está mais presente.

A decisão de executar ou não uma parte opcional deve ser tomada em função do nível de qualidade que o sistema pode fornecer ao usuário a partir dos recursos disponíveis. Uma questão em aberto é determinar se o mecanismo baseado no conceito de erro total do sistema, como um valor numérico obtido a partir da soma dos erros das tarefas individuais, é ou não capaz de expressar esta política satisfatoriamente. A atribuição de pesos às tarefas permite algum controle sobre o escalonamento das partes opcionais, porém limitado. Não existem na bibliografia trabalhos de Computação Imprecisa que utilizem o conceito de "especificação da variação na qualidade do sistema em sobrecarga" para dirigir o escalonamento em tempo de execução.

4.3 Proposta

A seção anterior citou aspectos ainda não completamente explorados com respeito à técnica de Computação Imprecisa. Nesta seção é descrita a proposta de trabalho para o restante do curso. É feita uma descrição do problema e seu contexto, seguida das questões básicas a serem abordadas.

4.3.1 Descrição do Problema e Seu Contexto

O objetivo do trabalho será mostrar como Computação Imprecisa pode ser usada na construção de aplicações distribuídas com requisitos de tempo real. Serão consideradas aplicações formadas por conjuntos de tarefas imprecisas. Nestas aplicações, a carga possui um componente limitado e conhecido em tempo de projeto. Este componente é formado por tarefas periódicas ou esporádicas, cujas propriedades temporais são todas conhecidas em tempo de projeto. A carga também pode possuir um componente dinâmico, desconhecido em tempo de projeto. Este componente é formado por tarefas criadas em tempo de execução ou por tarefas aperiódicas, sem intervalo mínimo entre ativações.

É suposto que a aplicação requer a garantia, em tempo de projeto, de que as partes obrigatórias das tarefas serão concluídas dentro dos respectivos deadlines. Obviamente, somente as tarefas pertencentes ao componente limitado e conhecido em tempo de projeto poderão possuir uma parte obrigatória não nula. Neste trabalho, o conceito de garantia dinâmica (chamado de "planning" por alguns autores [RAM 94]) não é aceito. Se existe a possibilidade de ocorrer a rejeição de uma tarefa em tempo de execução, ela deve ser considerada opcional para efeitos da especificação do sistema.

Com respeito ao subsistema de comunicação, serão supostas mensagens do tipo garantido. Estas mensagens terão um prazo de entrega garantido em tempo de projeto, desde que suas propriedades sejam também conhecidas em tempo de projeto. Não serão abordados no trabalho os protocolos de comunicação necessários para implementar tais mensagens. Os mecanismos do subsistema de comunicação serão abstraídos no trabalho. Apenas o seu serviço será usado.

4.3.2 Questões a Serem Abordadas

Para alcançar o objetivo proposto, será necessário resolver diversos problemas. São três as questões básicas a serem elucidadas: o modelo de tarefas, o suporte algorítmico e a validação do modelo através de exemplos significativos.

4.3.2.1 Proposição de um Modelo de Tarefas

A questão mais importante é a criação de um modelo de tarefas distribuída que incorpore técnicas de Computação Imprecisa. Em outras palavras, adaptar os conceitos associados com CI para ambientes distribuídos. Esta adaptação vai além do simples porte das técnicas para ambientes distribuídos. Deverão ser estudadas maneiras de aproveitar a execução distribuída na melhoria da própria técnica. Um exemplo de melhoria é a execução simultânea de várias versões da parte opcional de uma tarefa. Na criação deste novo modelo, deverão ser considerados as possíveis extensões citadas no início deste capítulo.

O aproveitamento dos conceitos e idéias citados dependerá da evolução dos trabalhos ligados ao desenvolvimento do modelo de tarefas. Embora algumas das extensões a serem estudadas também sejam úteis em um contexto inteiramente local, este trabalho visa especificamente sua aplicação em sistemas distribuídos.

Muitos modelos introduzidos na literatura apresentam limitações que dificultam a utilização dos mesmos em aplicações reais. O modelo a ser introduzido por este trabalho tem por objetivo diminuir estas limitações. Espera-se chegar a um modelo de tarefas que represente uma extensão dos modelos associados hoje com Computação Imprecisa. Em particular, qualquer extensão proposta deverá considerar o contexto de execução distribuída. Desta forma, será explorado o emprego de CI em modelos de tarefas mais flexíveis que os usados atualmente.

Os próximos parágrafos discutem alguns aspectos a serem considerados na elaboração de um modelo de tarefas que empregue Computação Imprecisa em ambiente distribuído.

Tipo de carga

Existem razões para o emprego de CI tanto em sistemas com carga limitada e conhecida em tempo de projeto quanto em sistemas com carga desconhecida em tempo de projeto. Um modelo de tarefas que se propõe a suportar o uso de Computação Imprecisa deve permitir os diversos tipos de carga onde a técnica é útil.

Primeiramente, considere a situação na qual toda a carga é limitada e conhecida em tempo de projeto, mas reservar recursos para o pior caso de todas as tarefas é inviável

devido ao custo. Ao mesmo tempo, a probabilidade de todas as tarefas apresentarem um comportamento de pior caso simultaneamente é pequena. Neste caso, a Computação Imprecisa permite transformar partes das tarefas em opcionais. O sistema é construido de maneira que exista uma reserva de recurso para todas as partes obrigatórias no pior caso. Já as partes opcionais são tratadas considerando-se o caso médio. Recursos são incluídos no sistema considerando-se apenas a necessidade média das partes opcionais. Em tempo de execução, dificilmente alguma parte opcional será sacrificada. Existem recursos para o caso médio e ainda existe a sobra dos recursos associados com as partes obrigatórias. Eles serão capazes de satisfazer pequenas sobrecargas. Neste caso, é importante que as sobras de recurso das partes obrigatórias, quando estas apresentam um comportamento mais favorável que o pior caso, possam ser aproveitadas. Existem formas de aproveitar as sobras tanto em executivos cíclicos ([FOH 94]) como em sistemas que trabalham com prioridades ([DAV 93a]).

Uma segunda situação de carga, diferente da primeira, acontece quando a carga não é conhecida em tempo de projeto. Obviamente não é possível garantir todas as tarefas (desconhecidas) do sistema. Uma opção nesta situação é dividir a carga em dois componentes. Um componente é formado por tarefas periódicas ou esporádicas, conhecidas em tempo de projeto. Desta forma, suas partes obrigatórias podem ser garantidas, através de um teste de escalonabilidade. O outro componente é composto por tarefas cujo comportamento é desconhecido em tempo de projeto. Ele é tratado, em tempo de execução, dentro da política do melhor esforço. É importante destacar que a Computação Imprecisa não impede a existência de tarefas puramente obrigatórias ou puramente opcionais. Não é necessário mudar nada na conceituação de CI para aceitar tarefas deste tipo. Assim, é possível a existência de um componente de carga composto por tarefas que possuem apenas a parte opcional.

Uma terceira situação de carga onde existem razões para usar Computação Imprecisa surge do fato dela permitir o emprego, em sistemas que devem ter um comportamento temporal garantido, de algorítmos com um tempo de execução no pior caso infinito ou extremamente grande. Para isto, deve ser utilizada a forma de programação baseada em múltiplas versões. A versão primária deve incluir o algorítmo preciso mas com duração imprevisível. A versão secundária deve conter um algorítmo capaz de produzir um resultado de menor qualidade, em um tempo limitado, no pior caso. Em tempo de execução, é estabelecido um limite de tempo para a versão primária. Se o resultado de qualidade maior puder ser obtido pela versão primária, dentro deste limite de tempo arbitrado, ele é usado. Caso contrário, a versão secundária é usada para que o sistema disponha de uma valor com qualidade minimamente aceitável. Se esta técnica for empregada em ambiente distribuído, é possível disparar as duas versões simultaneamente, diminuindo o tempo total de execução da tarefa.

Para que o modelo de tarefas proposto seja útil, ele deve ser capaz de acomodar estas diversas situações de carga. Nelas, Computação Imprecisa aparece como uma alternativa interessante para resolver dificuldades encontradas na construção de sistemas tempo real. O essencial é que o modelo de carga adotado não comprometa o funcionamento básico da técnica Computação Imprecisa:

- Teste de escalonabilidade, em tempo de projeto, para todas as partes obrigatórias;
- Melhor esforço, em tempo de execução, para todas as partes opcionais.