Um Modelo de Tarefas Distribuído Empregando Computação Imprecisa

Rômulo Silva de Oliveira* e Joni da Silva Fraga

Laboratório de Controle e Microinformática - LCMI-EEL-UFSC
Campus Universitário - Trindade - Florianópolis - SC
Caixa Postal 476 - CEP 88040-900
E-mail: romulo@lcmi.ufsc.br e fraga@lcmi.ufsc.br

Resumo

Sistemas computacionais de tempo real são identificados como aqueles submetidos a requisitos de natureza temporal. Uma técnica recente para o escalonamento neste tipo de sistema é a Computação Imprecisa. Neste trabalho é descrita uma proposta de uso da Computação Imprecisa no escalonamento de aplicações distribuídas com requisitos de tempo real. Um modelo de tarefas para ambientes distribuídos que incorpora conceitos de Computação Imprecisa é definido. É também apresentado um suporte algorítmico que atende os requisitos de natureza temporal da aplicação. Por fim, o modelo proposto será avaliado à luz da literatura existente.

Abstract

Real-Time Computing Systems are systems which have timing constraints. The Imprecise Computation technique has been proposed as an approach to the scheduling of Real-Time Systems. This paper describes the use of Imprecise Computation techniques in the scheduling of distributed real-time applications. It is introduced a task model suitable to distributed environments that incorporate the Imprecise Computation technique. Scheduling algorithms to satisfy the timing requirements of the application are also presented. The proposed model is compared to others present in the real-time literature.

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.

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. 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) do comportamento temporal.

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

---

* Professor do Instituto de Informática da UFRGS, em doutoramento no LCMI-UFSC.

exigidos. Esta técnica, conhecida pelo nome de Computação Imprecisa [LIU 94], flexibiliza o escalonamento tempo real.

Neste trabalho é introduzida uma proposta de uso da Computação Imprecisa no escalonamento de aplicações distribuídas com requisitos de tempo real. Um modelo de tarefas para ambientes distribuídos que incorpora conceitos de Computação Imprecisa é definido. É também apresentado um suporte algorítmico que atende os requisitos de natureza temporal da aplicação. Por fim, o modelo proposto será avaliado à luz da literatura existente na área.

O restante do texto está organizado da seguinte maneira: a seção 2 descreve o contexto onde este trabalho está inserido; a seção 3 define o modelo de tarefas proposto e discute os possíveis caminhos para solucionar o problema do escalonamento; na seção 4 são feitas considerações gerais sobre o modelo proposto; as conclusões finais aparecem na seção 5.

2. Contexto do Trabalho

Na maioria das vezes, sistemas de tempo real são descritos através de um modelo de execução baseado em tarefas ("tasks"). 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. 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.

A figura 1 mostra as 5 abordagens básicas para o escalonamento na construção de STR. Elas aparecem divididas em dois grupos, conforme o tipo de previsibilidade oferecida. O primeiro grupo de abordagens (garantia no projeto, figura 1) é aquele capaz de oferecer uma previsibilidade determinista. É empregado em sistemas onde é necessário garantir, em tempo de projeto, que todas as tarefas serão executadas dentro de seus deadlines. Obviamente, esta garantia é obtida a partir de um conjunto de premissas, ou seja, uma determinada hipótese de carga ("load hypothesis") e uma hipótese de faltas ("fault hypothesis").

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

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

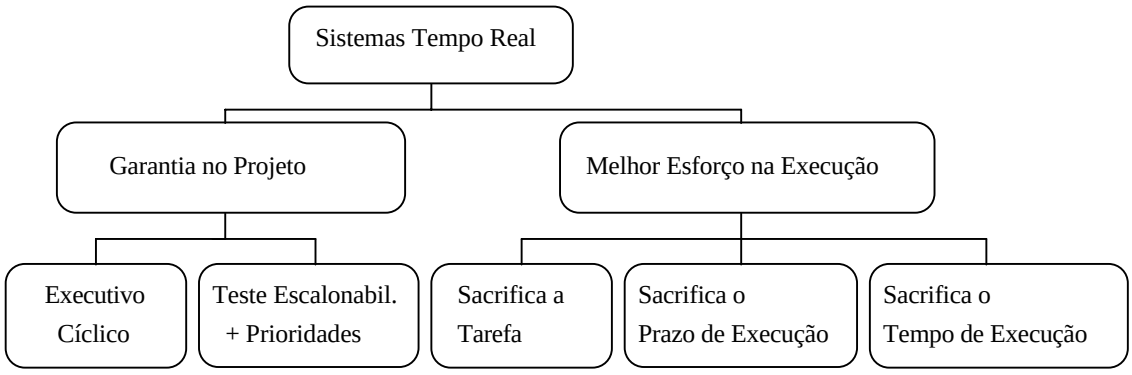

Figura 1 - Possíveis abordagens para sistemas tempo real.

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

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

Nas abordagens do segundo grupo (melhor esforço na execução, figura 1), não existe garantia, fornecida em tempo de projeto de que os deadlines serão cumpridos. O escalonamento tipo "melhor esforço" ("best effort") quando muito fornece uma previsibilidade probabilista sobre o comportamento temporal do sistema, a partir de uma estimativa da carga. Algumas propostas dentro desta linha oferecem uma "garantia dinâmica" ao determinar, em tempo de execução, quais prazos serão ou não atendidos.

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

2.1 Computação Imprecisa

As técnicas de Computação Imprecisa fazem parte das abordagens tipo "melhor esforço", com o sacrifício do tempo de execução das tarefas. Estas técnicas estão fundamentadas 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 parcialmente ou totalmente descartadas. Este mecanismo permite uma degradação controlada do sistema, na medida em que pode-se determinar o que não será executado em caso de sobrecarga. O mecanismo também permite a melhor utilização dos recursos pelas partes opcionais, que podem aproveitar recursos reservados em tempo de projeto e não utilizados pelas partes obrigatórias.

As técnicas de Computação Imprecisa como um todo são do tipo melhor esforço, pois não oferecem uma previsibilidade determinista para todas as tarefas do sistema. Entretanto, as partes obrigatórias tomadas isoladamente formam um subproblema a ser tratado pelas abordagens que oferecem garantias em tempo de projeto. Para isto, no conjunto das partes obrigatórias devem ser observadas as condições necessárias para a previsibilidade determinista, ou seja, carga limitada e conhecida e ainda recursos disponíveis para execução no pior caso.

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.

2.1.1 Formas de Programação da Computação Imprecisa e Função Erro

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 refina progressivamente o resultado da tarefa e será incluída como parte opcional. É a forma de programação que fornece maior flexibilidade ao escalonador. Algorítmos adaptáveis a este estilo de programação podem ser encontrados nas áreas de cálculo numérico, estatistica, pesquisa heurística, ordenação e consultas a banco de dados.

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. 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 função, em executa-la completamente ou não executa-la. Exemplos de aplicações que suportam este estilo de programação podem ser encontrados em processamento de sinal e processamento de imagem.

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.

Para efeitos de escalonamento, muitas propostas associam às tarefas valores de erro pela não execução de suas partes opcionais. Este valor de erro quantifica "a diferença entre a

qualidade do resultado preciso e a do resultado efetivamente gerado". Na maioria das propostas na literatura, este erro é suposto proporcional ao tempo de execução que faltou para concluir a parte opcional em questão. Outras propostas assumem restrições 0/1, ou seja, a parte opcional não executa e o erro é máximo ou a parte opcional executa completamente e o erro é zero. As funções erro dirigem o escalonamento de partes opcionais no sistema.

2.1.2 Revisão da Bibliografia

A maioria dos trabalhos que tratam de Computação Imprecisa encontrados na literatura consideram tarefas aperiódicas. Tipicamente, tarefas imprecisas são dinamicamente ativadas, durante a execução do sistema. Em tempo de execução, o algorítmo de escalonamento procura minimizar o erro associado com o 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. Dentro desta abordagem, as propostas apresentadas em [SHI 92], [HO 92b] e [HO 94] supõe uma programação através de funções monotônicas. Em [HO 92a] são consideradas tarefas com restrição 0/1, permitindo o uso de funções de melhoramento e múltiplas versões.

O problema de escalonar tarefas imprecisas periódicas é tratado em [CHU 90]. As partes obrigatórias são garantidas através da técnica conhecida como Taxa Monotônica ("rate monotonic"). Com respeito ao erro das partes opcionais, [CHU 90] classifica as tarefas periódicas em dois tipos. No primeiro tipo, o erro gerado em cada ativação vai sendo acumulado pela tarefa. No segundo tipo, o erro gerado em cada ativação da tarefa é independente das demais ativações desta mesma tarefa.

Os modelos de tarefas apresentados na literatura se limitam normalmente a tarefas independentes. Relações de precedência e as consequências da qualidade dos dados de entrada sobre as funções erro são ignorados. A única exceção é o trabalho apresentado em [LIU 94] onde o erro associado à uma tarefa é função da qualidade de seus dados de entrada e também de seu tempo de execução. O uso de técnicas de Computação Imprecisa em ambientes distribuídos não é tratado na literatura.

3. Modelo de Tarefas Distribuído Usando Computação Imprecisa

O objetivo deste trabalho é explorar o emprego da técnica Computação Imprecisa em ambiente distribuído. O escalonamento tempo real é geralmente resolvido em duas etapas no contexto de sistemas distribuídos. 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. O escalonamento local considera como carga as tarefas alocadas na primeira etapa.

A Computação Imprecisa é basicamente uma técnica para flexibilizar escalonamento local. Entretanto, o fato de tarefas locais incluídas em uma aplicação distribuída dependerem de eventos remotos, gera implicações que afetam o atendimento de restrições temporais associadas a estas tarefas. Logo, o fato da aplicação ser distribuída é relevante no uso desta técnica. Neste caso, a dependência de eventos remotos deve influir na qualidade de execução de tarefas locais.

3.1 Modelo de Tarefas Proposto

A nossa proposição de um modelo de tarefas distribuído implica na execução da aplicação em um sistema distribuído formado por um conjunto de processadores (ou nós), conectados através de um meio de comunicação. O envio de mensagens entre dois processadores quaisquer (comunicação remota) é delimitado por um tempo máximo Δ. O tempo para o envio de uma mensagem entre duas tarefas no mesmo processador será considerado zero para efeito de escalonamento. Na verdade, isto corresponde a incluir no tempo de computação da tarefa remetente o tempo necessário para o envio da mensagem.

Uma aplicação distribuída neste modelo é expressa na forma de um conjunto de atividades. A atividade é a entidade encapsuladora de tarefas que se comunicam e/ou se sincronizam. Cada atividade é representada por um grafo dirigido acíclico, correspondendo a um conjunto de tarefas. Os nodos desde grafo representam tarefas e os arcos representam relações de precedência. Uma relação de precedência tem origem em uma necessidade de sincronização e/ou passagem de dados entre duas tarefas de uma mesma atividade.

Este modelo parte da premissa que uma aplicação é formada por um conjunto $\boldsymbol{A}$ de atividades periódicas. Cada atividade $\boldsymbol{Ai}$, pertencente ao conjunto $\boldsymbol{A}$, corresponde a uma sequência infinita de ativações. As ativações ocorrem em intervalos regulares de tempo, caracterizando o aspecto periódico da atividade. A notação $\boldsymbol{Ai,j}$ será empregada para denotar a $\boldsymbol{j}$-ésima ativação da atividade $\boldsymbol{Ai}$, onde $\boldsymbol{Ai} \in \boldsymbol{A}$ e $\boldsymbol{1} \leq \boldsymbol{j} \leq \infty$. Uma atividade $\boldsymbol{Ai}$, pertencente ao conjunto $\boldsymbol{A}$, é caracterizada por um período $\boldsymbol{Pi}$, um deadline $\boldsymbol{Di}$, relativo ao início de cada ativação da atividade, e um peso $\boldsymbol{Wi}$, representando a importância funcional da atividade para o sistema. O deadline $\boldsymbol{Di}$ da atividade $\boldsymbol{Ai}$ está associada com a conclusão de todas as tarefas pertencentes a atividade $\boldsymbol{Ai}$. O deadline $\boldsymbol{Di}$ é sempre igual ou menor do que o período $\boldsymbol{Pi}$. É suposto que a primeira ativação de todas as atividades inicia no instante zero de tempo.

Toda aplicação $\boldsymbol{A}$ está associada também a um conjunto $\boldsymbol{T}$ contendo $\boldsymbol{N}$ tarefas, ou seja, a todas as tarefas pertencentes a todas as atividades da aplicação $\boldsymbol{A}$. As tarefas não possuem relações de exclusão mútua entre si. Qualquer tarefa pode ser preemptada, ou seja, ter sua execução suspensa e retomada mais tarde. Cada tarefa $\boldsymbol{Ti}$ possui associado um peso interno $\boldsymbol{Vi}$, o qual representa a importância funcional da tarefa dentro de sua atividade.

Cada tarefa $\boldsymbol{Ti}$, pertencente ao conjunto $\boldsymbol{T}$, é logicamente decomposta em uma parte obrigatória com tempo de execução máximo $\boldsymbol{Mi}$ e outra parte opcional com tempo de execução máximo $\boldsymbol{Oi}$. Estes tempos são tais que $\boldsymbol{Mi + Oi = Ci}$. A parte opcional somente pode iniciar sua execução após a conclusão da parte obrigatória.

3.2 Escalonamento

Embora uma solução de escalonamento precise considerar simultaneamente as partes obrigatórias e opcionais das tarefas, o objetivo do escalonamento como um todo é duplo. Primeiramente, o deadline das partes obrigatórias deve ser garantido. Em seguida, as partes opcionais devem ser escalonadas de maneira a minimizar o erro total observado no sistema. Para aumentar a clareza da apresentação, os dois aspectos serão considerados separadamente.

Nos últimos anos, resultados teóricos importantes foram obtidos a partir de algorítmos de escalonamento que trabalham com prioridades, especialmente prioridades fixas (abordagem Teste de Escalonabilidade + Prioridades, figura 1). O emprego desta técnica de escalonamento permite garantir, em tempo de projeto, que todas as partes obrigatórias serão executadas antes do respectivo deadline. A solução de escalonamento apresentada está baseada em prioridades fixas.

3.2.1 Alocação de Tarefas e Garantia das Partes Obrigatórias

Em tempo de projeto existe a preocupação de garantir os deadlines das partes obrigatórias. Em função disto, serão consideradas somente as partes obrigatórias das tarefas nos testes de escalonabilidade empregados em tempo de projeto. Em outras palavras, para qualquer tarefa ***Ti*** será suposto ***Oi*** igual a ***0*** e ***Ci*** igual ao valor ***Mi*** original.

O problema de escalonamento global do sistema é transformado, através dos passos descritos na sequência deste texto, em um conjunto de ***B*** problemas de escalonamento locais, sendo ***B*** o número de processadores no sistema. Cada tarefa é associada com um único processador específico. Não existe migração de tarefas em tempo de execução. O resultado da alocação deve ser tal que cada processador seja capaz de garantir a execução das partes obrigatórias antes dos respectivos deadlines. Cada tarefa ***Ti*** pertencente a uma atividade ***Aj*** tem o seu deadline individual ***DTi*** definido a partir do particionamento da folga existente na atividade, considerando-se os atrasos de comunicação entre processadores. Considere o exemplo mostrado na figura 2, onde aparece uma atividade com cinco tarefas.

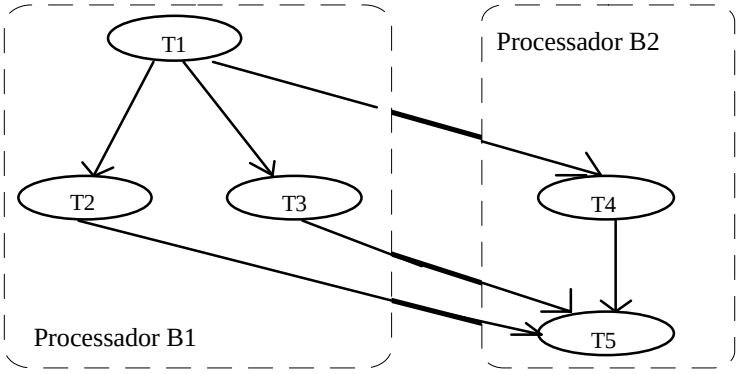

Figura 2 - Exemplo de uma atividade.

Uma tarefa ***Ti*** possui individualmente um valor mínimo ***DMINi*** e um valor máximo ***DMAXi*** para o seu deadline individual ***DTi***, ou seja, $DMINi \leq DTi \leq DMAXi$ **.** O valor mínimo ***DMINi*** deve ser tal que respeite o tempo de execução de ***Ti*** e de todas as tarefas que precedem ***Ti***. O valor máximo ***DMAXi*** deve ser tal que permita a execução de todas as tarefas que sucedem ***Ti*** antes do deadline da atividade.

Para efeito de escalonamento local, cada tarefa ***Ti*** deve ter um único deadline ***DTi***. Um dos problemas a serem resolvidos pelo algorítmo de alocação é determinar o ***DTi*** para cada tarefa ***Ti***. Observe que a alocação feita é importante, pois o atraso na rede deve ser considerado no momento de particionar a folga da atividade.

Outro aspecto a ser considerado é o *release jitter*. Para cada tarefa ***Ti*** da aplicação deve ser calculado o *release jitter* máximo ***Ji*** que ela pode sofrer em função de variação no tempo de execução das tarefas predecessoras. Este *release jitter* é consequência do fato dos tempos de computação e do atraso na rede serem valores máximos e não valores exatos. Uma discussão sobre o conceito de *release jitter* pode ser encontrada em [TIN 94b].

Em função da complexidade do problema de alocação, as soluções propostas na literatura tendem a ser subótimas. Em linhas gerais, podemos dividir as propostas em duas: pesquisa heurística e pesquisa aleatória. Na pesquisa heurística, o espaço de soluções é organizado na forma de uma árvore. O problema da alocação é resolvido através da identificação de um caminho nesta árvore. A identificação deste caminho é dirigida por uma heurística que procura sempre optar por alocações com melhores resultados. Quando fica claro

que o caminho escolhido não levará a uma solução, é feito um retorno ("backtracking") e outro caminho é testado.

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

Um método interessante é o recozimento simulado ("simulated annealing", [KIR 83], [TIN 92]). Enquanto 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". Neste trabalho optamos pelo uso da técnica do recozimento simulado no problema de alocação visando a garantia dos deadlines das tarefas, considerando só as partes obrigatórias. Para isto, é necessário adaptar a função que avalia a qualidade de uma solução. As restrições de alocação podem ser tratadas da mesma forma que são tratadas em [TIN 92].

Embora existam na literatura heurísticas para realizar o particionamento de folga em atividades ([KAO 94]), é possível utilizar o mesmo mecanismo do recozimento simulado. Neste caso, tanto a associação tarefa-processador quanto o deadline individual de cada tarefa são atribuidos aleatoriamente e o resultado avaliado. Para acelerar o processo de descarte de soluções inviáveis, os deadlines podem ser atribuidos dentro do intervalo ***DMIN*** e ***DMAX*** de cada tarefa.

Nos ***B*** problemas de escalonamento local, resultantes da alocação das tarefas aos processadores, é necessária a realização de testes de escalonabilidade. Definimos para a atribuição de prioridades às tarefas em cada um dos ***B*** problemas de escalonamento a política Deadline Monotônico [LEU 82]. Testes de escalonabilidade para esta técnica são discutidos em [AUD 91] e [AUD 93a]. Existem testes suficientes mas não necessários com complexidade $O(N^2)$, além de testes exatos com complexidade pseudo-polinomial. O impacto da existência de *release jitter* e relações de precedência entre tarefas causado sobre os testes de escalonabilidade associados com o Deadline Monotônico é analisado em [AUD 93b].

3.2.2 Escalonamento das Partes Opcionais

O emprego de Computação Imprecisa exige a definição de uma função de erro para as tarefas do sistema. Esta função de erro será utilizada, em tempo de execução, na escolha de quais partes opcionais serão sacrificadas. O conceito de erro para uma aplicação distribuída envolve 3 níveis, em uma forma hierarquizada.

Erro na tarefa

A forma mais aproximada da realidade seria construir uma curva "erro versus tempo de execução" específica para cada tarefa. Esta curva poderia ser obtida através de um conjunto de ensaios feitos com a tarefa isoladamente. O objetivo de tais ensaios seria obter uma curva média ("profile") para o comportamento temporal da tarefa. Entretanto, o uso de uma curva com formato qualquer torna o escalonamento muito custoso. Além disto, funções de erro contínuas exigem programação através de funções monotônicas, o que nem sempre é viável.

Com o objetivo de ampliar a aplicabilidade da técnica Computação Imprecisa, vamos considerar como formas de programação o uso de função melhoramento e de múltiplas (duas)

versões. Desta forma, as tarefas da aplicação apresentam uma restrição 0/1. A figura 3 ilustra a função de erro. Com a escolha de funções erro com restrição 0/1, a decisão de escalonar somente ***Mi*** ou ***Mi+Oi*** deve ser tomada antes do início da tarefa. Isto porque, no caso de múltiplas versões, depois de iniciada uma versão ela não pode ser abandonada. A escolha entre versão primária (***Mi+Oi***) e versão secundária (***Mi***) ocorre obrigatoriamente antes da execução da tarefa e esta decisão já embute a decisão de executar ou não ***Oi***.

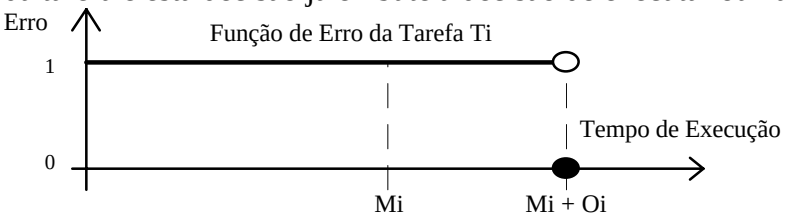

Figura 3 - Função de erro na forma de degrau.

Erro da atividade

O efeito dos erros em tarefas são percebidos a nível de suas atividades, dando origem ao conceito de atividade imprecisa. Da mesma forma que as atividades, cada tarefa ***Tk*** é ativada um infinito número de vezes. Suponha que ***Tk,j*** denote a ***j***-ésima ativação da tarefa ***Tk*** e que ***E(Tk,j)*** represente o erro associado com esta ativação ***Tk,j***, onde $Tk \in T$ e $1 \leq j \leq \infty$. O erro associado com a ***j***-ésima ativação da atividade ***Ai***, denotado por ***E(Ai,j)***, é dado pela soma ponderada dos erros de suas tarefas, ou seja:

$$E(Ai,j) = \sum_{Tk \in Ai} Vk \times E(Tk,j) \qquad \text{[Equação 3.1]}$$

Erro do sistema

Considere um intervalo de tempo qualquer ***[t1,t2)***, onde ***t2*** > ***t1***. Seja ***S[t1,t2)*** o conjunto de todas as ativações de atividades cujo deadline está dentro do intervalo de tempo ***[t1,t2)***. O erro total do sistema durante o intervalo ***[t1,t2)***, denotado por ***E[t1,t2)***, é definido como:

$$E[t1,t2) = \sum_{Ai,j \in S[t1,t2)} Wi \times E(Ai,j) \qquad \text{[Equação 3.2]}$$

Um algorítmo de escalonamento será considerado ótimo se ele obtiver o menor ***E[t1,t2)*** possível para qualquer intervalo ***[t1,t2)***.

A decisão de executar ou não a parte opcional de uma tarefa qualquer é tomada em tempo de execução. O processo de alocação definiu uma prioridade para cada tarefa. Localmente, em tempo de execução, a tarefa com prioridade maior recebe o processador. Sempre que uma tarefa inicia a execução é definido se somente a parte obrigatória vai executar ou se ambas as partes obrigatória e opcional vão executar. O fato desta decisão ser tomada no início de cada ativação da tarefa permite que a forma de programação múltiplas versões seja empregada. A inclusão da parte opcional de uma tarefa deve ser feita de tal forma que as garantias obtidas nos testes de escalonabilidade não sejam comprometidas.

Recentemente foi desenvolvida uma classe de algorítmos de escalonamento conhecidos como algorítmos de tomada de folga ("slack stealing") para uso em problemas de escalonamento que envolvem tarefas aperiódicas. Basicamente, estes algorítmos determinam o tempo máximo de processamento que pode ser tomado das tarefas periódicas garantidas, sem comprometer seus deadlines. O objetivo é aproveitar sobras no processador para executar

tarefas aperiódicas que não foram garantidas em tempo de projeto. A tomada de folga atende as tarefas aperiódicas usando qualquer tempo livre de processador que surja. Este é o caso do algorítmo apresentado em [LEH 92] que, em tempo de projeto, armazena as folgas identificadas em uma tabela. Este método é conhecido como método estático para tomada de folga.

Em [DAV 93] é apresentado um método dinâmico para tomada de folga que é adaptado para um modelo de tarefas que inclui sincronização entre tarefas, *release jitter* e tarefas esporádicas garantidas. Em [ELH 94] e [DAV 94] é mostrado como os algorítmos de tomada de folga podem ser empregados para garantir dinamicamente que tarefas aperiódicas podem ser executadas antes do respectivo deadline. Estes algorímos aproveitam a folga do sistema no momento da chegada da tarefa aperiódica para garantir sua execução.

Em [AUD 94] é citada a possibilidade de utilizar estes mesmos métodos de tomada de folga para escalonar partes opcionais de tarefas imprecisas. Neste caso, quando uma tarefa inicia sua execução, a folga do sistema no momento é usada para decidir se a parte opcional pode ou não ser garantida, respeitando o deadline da tarefa. Observe que esta garantia é obtida no início da execução da tarefa, o que respeita a restrição 0/1 e o emprego de múltiplas versões.

Uma vez definidas as premissas (função erro, precedência, etc) é possível então propor algorítmos de escalonamento. O enunciado então se resume a: "dadas as sobras de processador identificadas por um algorítmo de slack stealing, como escolher qual a parte opcional vai ocupar estas sobras?" A escolha deverá levar em consideração os erros potenciais das tarefas e um objetivo como a minimização da [Equação 3.2].

4. Considerações Gerais

Um aspecto importante é a influência das relações de precedência nas funções erro das tarefas. A questão básica em uma relação de precedência entre duas tarefas é: se a predecessora ***Ti*** da tarefa ***Tj*** não é executada completamente, qual o impacto deste fato sobre o erro gerado pela tarefa ***Tj***? A resposta depende da semântica da aplicação. Alguns cenários possíveis são:

- A relação de precedência reflete apenas uma sincronização e não o envio de dados. Neste caso, o erro de ***Ti*** não afeta ***Tj***.
- Existe dependência de dados na relação de precedência e o erro em ***Ti*** torna ***Tj*** menos efetiva, ou seja, ***Tj*** gasta mais tempo para chegar a um erro zero.
- O erro em ***Ti*** torna ***Tj*** mais importante, ou seja, o erro associado com a não execução da parte opcional de ***Tj*** fica maior.

Na proposta deste artigo partimos da premissa, mais simples, de que o erro em ***Ti*** não afeta sua tarefa sucessora ***Tj***. Está em nossos planos evoluir para situações onde possa ocorrer propagação de erro.

É importante salientar que devido a complexidade do problema de escalonamento como um todo, o custo de uma solução otimizada é proibitivo. Desta forma, a solução apresentada neste artigo é subótima em dois sentidos:

- Os algorítmos de alocação/escalonamento podem falhar em conseguir uma alocação que garanta as partes obrigatórias das tarefas, embora tal solução exista;
- O algorítmo de escalonamento local pode falhar em conseguir o menor erro possível para o sistema, no momento de escolher quais partes opcionais são sacrificadas.

A literatura é variável com respeito ao objetivo do escalonamento das partes opcionais. Também o contexto é geralmente diferente do contexto proposto neste trabalho. Estamos investigando algorítmos de escalonamento para as partes opcionais das tarefas que sejam adequados ao contexto de execução definido. Neste caso, o caminho adequado para avaliar os méritos dos algorítmos propostos é a simulação através da geração de carga sintética, execução dos algorítmos escolhidos e a obtenção de métricas compatíveis com os objetivos do escalonamento. A métrica principal é o erro total do sistema, definido pela [Equação 3.2]. Também é necessário considerar o impacto da alocação feita no escalonamento futuro de partes opcionais. Estas duas questões são atualmente objeto de estudo do nosso grupo.

5. Conclusão

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. Neste artigo foi proposto um modelo de tarefas para ambiente distribuído que incorpora a técnica de Computação Imprecisa. Também foi discutida a alocação e o escalonamento destas tarefas.

6. Referências

[AUD 91] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, may 1991.

[AUD 93a] N. C. Audsley, A. Burns, A. J. Wellings. Deadline Monotonic Scheduling Theory and Application. Control Engineering Practice, Vol. 1, No. 1, pp. 71-78, feb. 1993.

[AUD 93b] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Software Engineering Journal, Vol. 8, No. 5, pp.284-292, 1993.

[AUD 94] N. C. Audsley, A. Burns, R. I. Davis, A. J. Wellings. Integrating Best Effort and Fixed Priority Scheduling. Proc. of the 1994 Workshop on Real-Time Programming, 1994.

[CHU 90] J. Y. Chung, J. W. S. Liu, K. J. Lin. Scheduling Periodic Jobs that Allow Imprecise Results. IEEE Transactions on Computers, Vol.39, No.9, pp.1156-1174, sep. 1990.

[DAV 93] R. I. Davis, K. W. Tindell, A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. Proceedings of the IEEE Real-Time Systems Symposium, 1993.

[DAV 94] R. Davis. Guaranteeing X in Y: On-line Acceptance Tests for Hard Aperiodic Tasks Scheduled by the Slack Stealing Algorithm. Technical Report, Real-Time Systems Research Group, Department of Computer Science, University of York, 1994.

[ELH 94] C. McElhone. Adapting and Evaluating Algorithms for Dynamic Schedulability Testing. Department of Computer Science, University of York, february 1994.

[HO 92a] K. I.-J. Ho, J. Y.-T. Leung, W.-D. Wei. Scheduling Imprecise Computation Tasks with 0/1-Constraint. Technical Report, DCSE, University of Nebraska-Lincoln, 1992.

[HO 92b] K.I.-J. Ho, J.Y.-T. Leung, W.-D.Wei. Minimizing Constrained Maximum Weighted Error for Doubly Weighted Tasks. Technical Report, DCSE, Univ. of Nebraska-Lincoln, 1992.

[HO 94] K. I.-J. Ho, J. Y.-T. Leung, W.-D. Wei. Minimizing Maximum Weighted Error for Imprecise Computation Tasks. Journal of Algorithms, 16, pp. 431-452, 1994.

[JEN 85] E. Jensen, C. Locke, H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. Proc. of the Real-Time Systems Symposium, pp.112-122,1985.

[KAO 94] B. Kao, H. Garcia-Molina. Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks. Proc. Intern. Conf. on Distributed Computing Systems, pp.172-181, 1994.

[KIR 83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. Optimization by Simulated Annealing. Science (220), pp. 671-680, 1983.

[KOP 89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabi, C. Senft, R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. IEEE Micro, pp. 25-40, february 1989.

[LEH 92] J. P. Lehoczky, S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. Proceedings of the IEEE Real-Time Systems Symposium, pp. 110-123, 1992.

[LEU 82] J. Y. T. Leung, J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. Performance Evaluation, 2 (4), pp. 237-250, december 1982.

[LIU 94] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung. Imprecise Computations. Proceedings of the IEEE, Vol. 82, No. 1, pp. 83-94, january 1994.

[RAM 89] K. Ramamritham, J. A. Stankovic, W. Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. IEEE Transactions on Computers, Vol.38, No.8, pp.1110-1123, august 1989.

[SHA 94] L. Sha, R. Rajkumar, S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. Proceedings of the IEEE, Vol. 82, No. 1, pp. 68-82, january 1994.

[SHI 92] W.-K. Shih, J. W. S. Liu. On-line Scheduling of Imprecise Computations to Minimize Error. Proceedings of IEEE Real-Time Systems Symposium, pp. 280-289, 1992.

[TIN 92] K. W. Tindell, A. Burns, A. J. Wellings. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. Real-Time Systems, Vol. 4, No. 2, pp. 145-165, june 1992.

[TIN 94a] K. W. Tindell, A. Burns, A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. Real-Time Systems, pp. 133-151, 1994.

[TIN 94b] K. W. Tindell. Fixed Priority Scheduling of Hard Real-Time Systems. Department of Computer Science thesis, University of York, 1994.