View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Institucional da UFSC

Universidade Federal de Santa Catarina
Programa de Pós-Graduação em Engenharia Elétrica

Escalonamento de Tarefas Tempo Real com Controle de *Valor* em Situações de Sobrecarga

Dissertação submetida à Universidade Federal de Santa Catarina
como requisito parcial à obtenção do grau de

Mestre em Engenharia Elétrica

por

Crineu Tres

Florianópolis, Abril de 2006

Escalonamento de Tarefas Tempo Real com Controle de *Valor* em Situações de Sobrecarga

Crineu Tres

Esta dissertação foi julgada adequada para a obtenção do título de **Mestre em Engenharia** na especialidade **Engenharia Elétrica**, área de concentração **Controle, Automação e Informática Industrial**, e aprovada em sua forma final pelo curso de Pós-Graduação.

Florianópolis, Abril de 2006.

Prof. Dr. Leandro Buss Becker

Prof. Dr. Alexandre Trofino Neto
Coordenador do curso de Pós-Graduação em Engenharia Elétrica
da Universidade Federal de Santa Catarina.

Banca Examinadora

Prof. Dr. Leandro Buss Becker, orientador

Prof. Dr. Rômulo Silva de Oliveira

Prof. Dr. Carlos Eduardo Pereira

Prof. Dr. Carlos Barros Montez

Prof. Dr. Antônio Augusto Fröhlich

Resumo

Aplicações tempo real modernas são dinâmicas, e não podem basear-se em cargas de trabalho no pior caso para oferecer garantias de execução. Então são necessários algoritmos de escalonamento capazes de tratar situações onde não existem recursos suficientes para todo o sistema. Nesse contexto a teoria do escalonamento baseado em valor *torna-se útil para adicionar generalidade e flexibilidade a tais sistemas. Esta dissertação apresenta um estudo comparativo entre o comportamento de diferentes escalonadores tempo real em situações de sobrecarga, considerando o papel desempenhado pelo parâmetro* valor*. Os algoritmos analisados são EDF, HVF, HDF e DMB (*Dynamic Misses Based*). Este último é introduzido aqui para alterar dinamicamente o* valor *das tarefas refletindo suas importâncias de acordo com o número de* deadlines *perdidos. O principal objetivo da análise é definir o algoritmo de escalonamento mais adequado para ser usado em conjunto com a estratégia de escalonamento TAFT (*Time-Aware Fault-Tolerant*), levando-se em conta sua capacidade de utilizar o parâmetro* valor *para controlar o comportamento das tarefas. Os resultados obtidos mostram que algoritmos de escalonamento que usam o* valor *apresentam um melhor desempenho geral, com a penalidade da diminuição da funcionalidade. O algoritmo DMB aliado ao TAFT alcançou os resultados mais promissores devido à sua capacidade de controlar a degradação das tarefas durante a execução da aplicação.*

Abstract

Modern real-time applications are very dynamic, and cannot cope with worst case execution workload (e.g. to avoid overload situations). Therefore, scheduling algorithms that are able to deal with situations where there are no sufficient resources for the whole system are required. Within this context, the value-*based scheduling appears as an interesting approach to add generality and flexibility to such systems. This dissertation presents a comparative study among different real-time schedulers analyzing their behavior during overload conditions, considering the impact with the use of the* value *parameter. The evaluated algorithms are EDF, HVF, HDF, and DMB (Dynamic Misses Based). The later was introduced here to dynamically change tasks* value *reflecting their importance according to the number of deadline misses. The main goal of the performed evaluation is to define the most suitable scheduling algorithm to be used in conjunction with the TAFT (Time-Aware Fault-Tolerant) scheduler, taking into consideration their capacity to use the* value *parameter to control tasks behavior. Obtained results show that the scheduling algorithms that use the* value *present better overall performance, with the penalty of some functionality loss. However, in conjunction with TAFT the DMB algorithm reached the most promising results because of its ability to control tasks degradation in a gracefully way along execution.*

Sumário

Resumo iii

Abstract iv

Lista de Figuras vii

Lista de Abreviaturas viii

1 Introdução 1

1.1 Motivação . . . 2
1.2 Objetivos . . . 3
1.3 Organização do Trabalho . . . 3

2 Trabalhos Relacionados 5

2.1 Escalonamento Tempo Real . . . 5
2.2 O Parâmetro *Valor* . . . 6
2.2.1 Definição Original de *Valor* . . . 6
2.2.2 Uso do *Valor* para Tomadas de Decisões . . . 8
2.2.3 Extensão do Uso de *Valor* . . . 9
2.2.4 *Valor versus Deadline* . . . 10
2.2.5 Flexibilidade Através do *Valor* . . . 12
2.2.6 Escolha do Valor . . . 13
2.2.7 Exemplos do Uso de *Valor* . . . 14
2.2.8 Estado da Arte . . . 16
2.3 Política de escalonamento TAFT . . . 18
2.3.1 Escalonamento Segundo o Modelo de *TaskPairs* . . . 18
2.3.2 Conceito TAFT de Escalonamento . . . 21
2.3.3 Determinação dos Tempos de Execução das *MainParts* . . . 24
2.3.4 Mecanismo de Escalonamento do TAFT . . . 25
2.3.5 Extensões do Conceito TAFT . . . 28
2.4 Resumo . . . 29

3 Definição do Problema **31**
3.1 Descrição Informal . . . 31
3.2 Critérios de Avaliação dos Algoritmos . . . 33
3.3 Algoritmos de Escalonamento Avaliados . . . 34
3.3.1 Algoritmo *Dynamic Misses Based* . . . 35
3.4 Variação da Carga do Sistema . . . 38
3.5 Metodologia Utilizada . . . 39

4 Projeto de Experimentos para Simulação **40**
4.1 *Benchmark* para Sistemas Tempo Real . . . 40
4.2 Ferramenta de Simulação . . . 43
4.2.1 Modelagem . . . 44
4.2.2 Coleta de Estatísticas . . . 47
4.2.3 Números Aleatórios . . . 47
4.2.4 Funções de Distribuição de Probabilidades . . . 48
4.2.5 Conjunto de Tarefas . . . 49
4.2.6 Carga do Sistema . . . 51
4.3 Considerações Finais . . . 52

5 Resultados Obtidos **54**
5.1 Avaliação dos Algoritmos com *Valores* Constantes . . . 54
5.1.1 Algoritmos Tradicionais . . . 55
5.1.2 Escalonador TAFT . . . 58
5.1.3 Resumo . . . 62
5.2 Avaliação do DMB-TAFT com *Valores* Dinâmicos . . . 63
5.2.1 Modos Definidos de Execução . . . 65
5.2.2 Mecanismo de Predição . . . 66
5.2.3 Definição de uma Tarefa Crítica . . . 67
5.2.4 Síntese do Uso de *Valores* Dinâmicos . . . 68

6 Conclusões e Trabalhos Futuros **70**

Referências Bibliográficas **73**

Lista de Figuras

1.1 Exemplo da variação do tempo de execução na leitura de um sensor [1] 2

2.1 Exemplos de *Time/Utility Functions* 7
2.2 Modelo de bloco *try-except* [23] 19
2.3 Modelo de execução de um *TaskPair* [23] 20
2.4 Exemplo de escalonamento de *TaskPairs* [26] 20
2.5 Curva da PDF para estimação do parâmetro C e soma das probabilidades em determinado instante [29] 25
2.6 Estrutura de um *TaskPair* segundo o TAFT [29] 26

4.1 Interface gráfica do software OMNeT++ 43
4.2 Diagrama de classes simplificado do simulador construído 45
4.3 Distribuição de valores segundo a PDF $uniforme$ 48
4.4 Distribuição de valores segundo a PDF $beta(2, 3)$ 49
4.5 Diferentes comportamentos das duas séries de tarefas utilizadas 50
4.6 Diferenças entre utilizações nominal e efetiva 52

5.1 Desempenho geral dos escalonadores através da UA_R 55
5.2 Comportamento individual das tarefas: EDF 56
5.3 Comportamento individual das tarefas: DMB 57
5.4 Comportamento individual das tarefas: HDF 57
5.5 Comportamento individual das tarefas: HVF 58
5.6 Desempenho geral sob a política TAFT através da UA_R 59
5.7 Comportamento individual das tarefas: EDF em conjunto com o TAFT 60
5.8 Comportamento individual das tarefas: HDF em conjunto com o TAFT 60
5.9 Comportamento individual das tarefas: HVF em conjunto com o TAFT 61
5.10 Comportamento individual das tarefas: DMB em conjunto com o TAFT 61
5.11 Utilização efetiva e percentual de execuções de *ExceptionParts* 64
5.12 Simulação segundo dois modos definidos de execução 65
5.13 Simulação através do uso de um componente monitor 67
5.14 Simulação considerando-se uma tarefa crítica 67

Lista de Abreviaturas

API	*Application Program(ming) Interface*
AVTD	*Adaptive Value Density Threshold*
CPU	*Central Processing Unit*
DASA	*Dependent Activity Scheduling Algorithm*
DMB	*Dynamic Misses Based*
ECET	*Expected Case Execution Time*
EDF	*Earliest Deadline First*
EDL	*Earliest Deadline as Late as possible*
EP	*ExceptPart* ou *ExceptionPart*
FIFO	*First In, First Out*
HDF	*Highest Density First*
HVF	*High Value First*
LRT	*Latest Release Time*
MP	*MainPart*
OCET	*Optimistic Case Execution Time*
PDF	*Probability Distribution Function*
RED	*Robust Earliest Deadline*
RNG	*Random Number Generator*
TAFT	*Time-Aware Fault-Tolerant*
TP	*TaskPair*
TPS	*TaskPair-Scheduling Model*
TUF	*Time/Utility Function*
UA	*Utility Accrued*
UA_R	*Utility Accrued Ratio*
VCF	*Variable Cost Functions*
WCET	*Worst Case Execution Time*

Capítulo 1

Introdução

Sistemas computacionais tempo real enfrentam requisitos que são inerentemente dinâmicos, e uma estratégia para assegurar a execução da aplicação deve ser utilizada. A garantia de cumprimento dos *deadlines* em sistemas tempo real críticos é comumente obtida através da análise das tarefas com base no tempo de execução de pior caso (*Worst Case Execution Time* - WCET). Essa prática, embora confiável, é bastante pessimista, uma vez que o tempo médio de execução de uma tarefa é geralmente menor do que o tempo no pior caso, e recursos previamente alocados permanecem então ociosos.

Apesar desse pessimismo, o WCET é necessário e utilizado em testes de aceitação, e é a garantia indispensável para a grande maioria dos algoritmos de escalonamento tempo real. De fato essa garantia é um pouco extrema, e leva a situações da chamada "sobrecarga artificial", onde teoricamente o sistema deveria estar com seus recursos em plena utilização mas, na prática, apresenta uma utilização bem abaixo do esperado. Nesses casos novas tarefas não podem ser adicionadas, embora existam recursos disponíveis, pois a garantia de execução seria violada.

A figura 1.1 representa um exemplo do pessimismo gerado com o uso do WCET. O gráfico mostra o tempo necessário para execução da tarefa de processamento dos dados de um determinado sensor em um dispositivo móvel. É possível ver que apenas 5% das tarefas executaram acima de 60ms, sendo que este valor está bem abaixo do WCET medido, que é de aproximadamente 140ms.

Sem fazer uso da garantia gerada pelo emprego do WCET, um sistema tempo real torna-se suscetível a situações de sobrecarga, onde os recursos disponíveis são insuficientes e precisam ser alocados através de um processo de escolha. Sob essas condições, algoritmos de escalonamento tradicionais geralmente exibem um péssimo desempenho, mostrando-se incapazes de realizar uma diferenciação entre as tarefas que devem ser descartadas e aquelas que devem ser executadas.

Da mesma forma, sistemas tempo real modernos não possuem recursos em

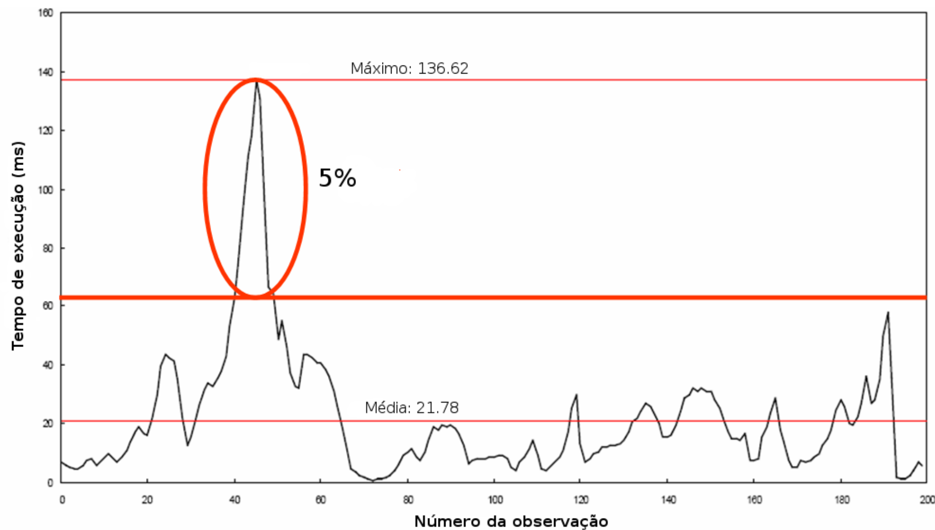


Figura 1.1: Exemplo da variação do tempo de execução na leitura de um sensor [1]

abundância a ponto de recair no uso de cargas de trabalho baseadas em WCET para que sua execução seja garantida. Várias alternativas foram desenvolvidas no intuito de aumentar a utilização efetiva de um sistema tempo real crítico e dinâmico. Nesse trabalho nos concentraremos na política de escalonamento *Time-Aware Fault-Tolerant* (TAFT), que substitui o uso de WCET nas tarefas pelo tempo esperado de execução (*Expected Case Execution Time* - ECET). Por esse motivo sobrecargas transientes são esperadas com mais freqüência, e um mecanismo de tratamento de exceção é oferecido para estas situações.

Embora desempenhe um papel chave no objetivo de aumentar a utilização efetiva, o escalonador TAFT também sofre de um problema comumente verificado em algoritmos tradicionais trabalhando em sobrecargas: incapacidade de distinção entre importância e urgência das tarefas, e o conseqüente fraco desempenho nesses casos. É justamente esse problema que o parâmetro *valor* tenta sanar.

1.1 Motivação

O significado da "importância" de uma tarefa vem sendo amplamente discutido durante os últimos anos. Jensen em [2] introduz o parâmetro *valor* com o objetivo de permitir ao escalonador o controle da execução das tarefas especificamente em situações de sobrecarga. A proposta atrela *Value Functions* a tarefas, e essas funções definem o quanto de utilidade o término da tarefa representa ao sistema. O escalonamento é realizado através de métodos que consideram o parâmetro *valor*, de modo que ao final da execução a máxima utilidade total possível seja adquirida.

Trabalhos realizados nessa área, como o de McElhone [3], demonstraram que o uso do parâmetro *valor* consiste em uma maneira simples de adicionar generalidade

e flexibilidade ao sistema utilizado. No entanto o uso de *Value Functions* exige um grande poder de processamento durante a fase de escalonamento, como atestam Tokuda em [4] e Wendorf em [5], o que acarreta a inviabilidade da adoção dessa técnica por qualquer tipo de sistema tempo real - já que em alguns casos pode ser necessário o uso de um *hardware* específico para esse encargo. Além disso, a teoria do escalonamento baseado em *valor* é altamente dependente da escolha propriamente dita desse *valor*, como afirma Burns em [6], haja vista que os benefícios inerentes dessa técnica são anulados por escolhas erradas ou inadequadas desses *valores*.

Dentre as motivações deste trabalho, salienta-se a necessidade existente em aplicações altamente dinâmicas, como os sistemas móveis cooperativos, de diferenciar a importância de uma tarefa da sua urgência. Como exemplos de sistemas que podem se beneficiar desta questão destacam-se as aplicações móveis cooperativas, apresentadas por Schemmer em [1] - particularmente o futebol de robôs. Em especial, a carência por mecanismos deste tipo é enfatizada por Becker em [7]. Por esta razão, este trabalho visa a integração do parâmetro *valor* junto à política de escalonamento TAFT, conforme detalhado na próxima seção.

1.2 Objetivos

O objetivo principal desta dissertação é a avaliação e posterior incorporação do comportamento do parâmetro *valor* na política de escalonamento TAFT, além do desenvolvimento de um método para controle do comportamento de tarefas em situações de sobrecarga. De modo mais específico, pretende-se dar continuidade a trabalhos anteriores nesta área, que contêm soluções insatisfatórias para definir a importância de uma tarefa e, portanto, sugerem a adoção de um novo mecanismo que desempenhe essa função.

Através da simulação de sistemas tempo real objetiva-se recriar experimentos prévios, adicionando melhorias e novas técnicas em busca de seu aperfeiçoamento. Além disso, é pretendida também a criação de um novo método de designação de prioridades em situações de sobrecarga, de maneira que o comportamento do sistema a longo termo seja equilibrado, com tarefas apresentando taxas de perdas de *deadlines* balanceadas, e, preferencialmente, controláveis.

1.3 Organização do Trabalho

O restante do trabalho é organizado da seguinte forma. O capítulo 2 apresenta um resumo histórico da criação e uso do parâmetro *valor* e do conceito de escalonamento TAFT, e ao final mostra o estado da arte atual das duas áreas; a descrição do problema

encontrado e a especificação dos conceitos e métodos utilizados durante a comparação dos algoritmos existentes com o novo modelo proposto são feitas no capítulo 3; o capítulo 4 descreve em detalhes o ambiente de simulação utilizado durante a fase de experimentação, assim como a definição das variáveis de configuração do ambiente; resultados da simulação são postos lado a lado e comparados no 5, segundo as métricas e critérios definidos, com a finalidade de decidir qual a melhor estratégia a ser utilizada; por fim, o capítulo 6 sintetiza as conclusões atingidas e propõe trabalhos futuros a serem realizados nessa área.

Capítulo 2

Trabalhos Relacionados

Neste capítulo é efetuada a revisão bibliográfica necessária para a realização do trabalho proposto. Primeiramente apresenta-se um resumo da teoria geral sobre escalonamento tempo real, onde procura-se citar as principais referências bibliográficas relacionadas. O restante do capítulo é dividido em duas seções principais: a primeira delas discute a criação e evolução do parâmetro *valor* em sistemas de escalonamento tempo real; posteriormente o conceito de escalonamento *Time-Aware Fault-Tolerant* (TAFT) é apresentado e descrito em detalhes. Por fim, apresenta-se um resumo com as principais contribuições levantados ao longo do capítulo e que fazem parte do escopo desta dissertação.

2.1 Escalonamento Tempo Real

A área de Escalonamento Tempo Real possui uma bibliografia variada e sólida. Liu e Wayland em [8] dissertam sobre a otimalidade de escalonadores de prioridade fixa em ambiente tempo real crítico[1], e demonstram suas propriedades e limitações matemáticas.

Chetto e Chetto em [9] apresentam um estudo profundo dos algoritmos de escalonamento do tipo *Earliest Deadline*, tanto *Earliest Deadline First* (EDF) como *Earliest Deadline as Late as possible* (EDL), incluindo provas matemáticas de suas propriedades e teoremas.

Piñedo em seu livro [10] escreve sobre teoria e aplicações de algoritmos de escalonamento, incluindo definições de modelos determinísticos e estocásticos, teoria da complexidade e classificação de cada modelo, heurísticas, problemas clássicos e métodos de construção e implementação. Embora seja uma referência bastante geral, o mesmo também abrange a área de escalonamento tempo real.

O livro "Real-Time Systems" [11] de Liu engloba grande parte da literatura clássica

[1] *Hard real-time.*

e é uma referência na área. Apresenta um conteúdo bastante amplo, que vai desde definições de termos básicos e diferentes métodos de escalonamento até o uso de recursos compartilhados, sistemas multiprocessados, comunicação tempo real e sistemas operacionais tempo real.

Em [12] Farines et al. discorrem especificamente sobre sistemas tempo real, suas características e funcionalidades. Sha et al. em [13] fazem uma retrospectiva bastante resumida sobre aspectos importantes e pontos chave da teoria do escalonamento em sistemas tempo real e aplicações.

A fundamentação teórica deste trabalho, referente a essas áreas, é baseada nas literaturas acima citadas, e elas podem ser consultadas para maiores detalhes.

2.2 O Parâmetro *Valor*

Esta seção apresenta um resumo de trabalhos sobre escalonamento tempo real relacionados ao parâmetro *valor*, organizados cronologicamente. A seção começa com sua formulação original, passa por diversos estudos e as respectivas conclusões alcançadas e termina com a exibição do estado da arte atual dessa área.

As primeiras menções sobre escalonamento baseado em *valor* foram feitas por Gouda et al. em um trabalho realizado para o governo e não publicado [14]. A aplicação consistia de um radar de defesa antimíssil com um poder de processamento muito alto, porém com eficiência bastante aquém do esperado. O objetivo final, que consistia basicamente em aumentar essa eficiência, foi alcançado com sucesso. Entretanto a solução não foi implementada e o trabalho nunca chegou a se tornar público.

2.2.1 Definição Original de *Valor*

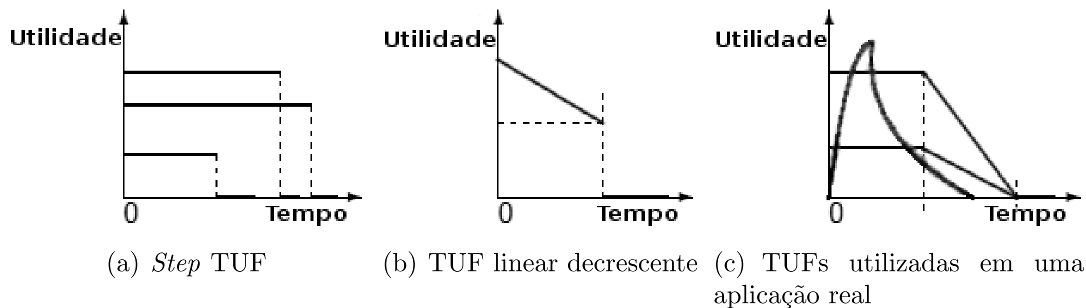
A primeira publicação sobre o tema foi feita por Jensen em [2], e define o conceito de *valor* em escalonamento tempo real. Segundo ele o término de um processo, ou um conjunto de processos, em um sistema tempo real representa um ganho, tendo, portanto, um *valor*, que pode ser expresso em função do tempo. O *valor* pode ser usado também como uma ferramenta para medir a eficiência dos escalonadores utilizados em sistemas tempo real sendo este o primeiro passo para o desenvolvimento de novas políticas que escalonem processos de modo a maximizar explicitamente esse *valor* (ou ganho) coletivo do sistema.

Jensen lança a tese de que o *valor*, o qual é variável em função do tempo, apresentado ao término de um processo é a principal diferença entre sistemas de tempo real e outros sistemas computacionais. O argumento é que, embora o tempo seja importante em todo tipo de sistema, nos sistemas tempo real é visto como uma parte crucial da correção

lógica[2] da aplicação.

É ressaltado também o fato de que escalonadores comuns não conseguem apresentar um comportamento adequado durante situações de sobrecarga, sem controle definido de quais processos devem ter os *deadlines* postergados, acarretando falhas difíceis de serem previstas e prejudicando a confiabilidade e a manutenabilidade[3] do sistema.

A fim de se avaliar o resultado do escalonamento é atribuído a cada processo um parâmetro denominado *valor*. Esse parâmetro é expresso em função do tempo, ou seja: é definido o *valor* do término do processo para o sistema, a qualquer hora. Então o sistema é "recompensado" com o montante determinado por essa função ao final de cada execução do processo. Esse tipo de função é chamado de *Time/Utility Function* (TUF)[4]. Um conjunto de processos é definido e a soma dos *valores* obtidos serve como métrica de desempenho dos algoritmos sob diversas condições de carga. A figura 2.1 exibe alguns exemplos de TUFs.



(a) *Step* TUF (b) TUF linear decrescente (c) TUFs utilizadas em uma aplicação real

Figura 2.1: Exemplos de *Time/Utility Functions*

É importante observar que uma TUF decompõe importância e urgência de uma tarefa. Enquanto a urgência é representada pelo *deadline*, medido no eixo x, a importância é denotada pelo *valor* da tarefa, através do eixo y.

Durante o processo de comparação de desempenho, além de utilizar alguns algoritmos já presentes na literatura como o *Earliest Deadline First* (EDF) [8] e o tradicional *First In, First Out* (FIFO), Jensen introduz novos algoritmos motivados pelo uso do parâmetro *valor* no processo de escalonamento. Um dos algoritmos introduzidos é o *High Density First* (HDF)[5], que escalona tarefas por ordem da densidade V/C, onde V representa o *valor* e C o tempo de processamento das tarefas.

Ao final do estudo Jensen chega a algumas conclusões intuitivas, declarando que em situações com pouca carga na CPU (*underload*) é inútil tentar superar o algoritmo EDF em desempenho, visto que ele é ótimo. Os testes também atestaram que os algoritmos

[2] *Correctness.*

[3] *Maintainability.*

[4] Na realidade esse trabalho referencia esse tipo de função pelo nome original de *Value Functions*. Mais tarde essas funções chegaram a ser citadas como *Benefit Functions*, a fim de evitar confusões. Contudo, firmaram-se na literatura e são atualmente conhecidas como *Time/Utility Functions*.

[5] Introduzido neste trabalho com o nome de *Value Density Schedule*.

criados superam aqueles existentes até então em situações de sobrecarga, avaliando assim a métrica pré-estabelecida.

2.2.2 Uso do *Valor* para Tomadas de Decisões

Em sua tese de doutorado [15] Locke se concentra no problema de decisão envolvido na escolha de uma lista de processos que devem ser executados em um sistema tempo real, e se propõe a desenvolver uma nova metodologia para tratar esta questão.

Como argumento inicial são citados vários procedimentos, normalmente seguidos em um processo de escolhas, e, em seguida, as diferenças encontradas durante esse mesmo processo efetuado em um sistema tempo real. Podemos usar como exemplo dessas diferenças o fato de que normalmente a coleta de todas as informações necessárias a uma decisão ótima é necessária, porém em sistemas tempo real geralmente não há tempo hábil para que isso aconteça.

A motivação do trabalho é o fato de que freqüentemente em sistemas computacionais as informações presentes são incompletas ou imprecisas, os recursos disponíveis são insuficientes, o algoritmo para solução do problema é intratável e, mesmo assim, uma decisão deve ser tomada. É por isso que deve ser feito o melhor esforço (*Best Effort*) para a tomada de decisões.

De acordo com Locke, entende-se por *Best Effort Decision Making* como sendo a tomada de decisões em um ambiente no qual a melhor decisão possível deva ser feita, sem levar em consideração a qualidade ou suficiência dos dados de entrada. Locke utiliza o conceito introduzido por Jensen de separação entre sistemas tempo real e outro tipo de sistemas pelo *valor* dos processos, e procura incluir o parâmetro *valor* na análise dos resultados.

Além da criação de heurísticas necessárias para o escalonamento, outros objetivos da tese são: (1) fazer essas heurísticas aplicáveis a sistemas multiprocessados com memória compartilhada; (2) construir um simulador flexível que permita a avaliação completa do algoritmo produzido; (3) comparar com outros escalonadores em sistemas já existentes, com as melhores decisões tomadas por humanos em situações críticas e com um limite máximo de *valor* obtido por um conjunto particular de tarefas e (4) determinar como um escalonador desse tipo pode ser usado em sistemas reais.

Uma das premissas mais importantes definidas durante o processo de desenvolvimento do algoritmo é que ele não irá, necessariamente, dividir o processador com os processos que estão sendo escalonados. Isso significa que o esforço computacional não é levado em conta durante a construção do algoritmo, e a sobrecarga (*overhead*) apresentada por ele é ignorada pelos resultados obtidos ao longo da fase de simulação.

Os principais resultados obtidos por Locke foram: (1) a criação de um algoritmo bastante geral, que se propõe a escalonar tarefas tendo como parâmetro principal as

TUFs e (2) a análise das implicações que um escalonador desse tipo traz às definições e implementações usadas atualmente em um sistema com um conjunto de políticas de escalonamento definidas pelo usuário.

Ao estender o uso de conceitos do escalonamento baseado em *valor* para áreas práticas esta tese transformou-se em uma referência sólida na área e é bastante citada por outras bibliografias, principalmente quando há uso de métodos do tipo melhor esforço.

2.2.3 Extensão do Uso de *Valor*

A tese de doutorado de Clark [16] propõe as seguintes contribuições para a área de tempo real: um método formal para análise de algoritmos de escalonamento; o *Dependent Activity Scheduling Algorithm* (DASA), que integra compartilhamento de recursos com métodos tradicionais de escalonamento, específico para sistemas de controle supervisório; e um simulador, onde os testes que atestaram o desempenho do DASA foram realizados.

Os principais objetivos que o algoritmo DASA se propõe a alcançar são:

- minimizar esforço – aplicar o mínimo de esforço para efetuar o escalonamento, possivelmente utilizando aborções para apressar o processo;
- maximizar retorno (ou benefício) – examinar tarefas em ordem decrescente de densidade de *valor* potencial[6], dessa forma sempre obtendo o maior retorno (como definido pela aplicação) pelo investimento (de tempo computacional);
- maximizar as chances de um *deadline* ser cumprido;

Ao final do trabalho é argumentado que a maioria desses objetivos foi, de uma forma ou outra, satisfeita.

Ao contrário de Locke, Clark leva em consideração o *overhead* presente no algoritmo. Embora a análise tenha sido feita através do próprio simulador construído, é admitido que esse *overhead* possa tornar o algoritmo inviável para um nicho específico de sistemas tempo real.

É concluído que utilizando-se de restrições temporais, importância descrita através de funções e dependências entre tarefas, sistemas de controle supervisório podem ser eficientemente escalonados. O algoritmo DASA, concebido neste trabalho, obteve bons resultados para os objetivos traçados, embora não seja adequado para qualquer tipo de sistema tempo real. Ao final é salientado que o uso de *valor* torna factível a distinção entre importância e urgência de uma atividade.

[6] *Value Density* é uma medida de quanto *valor* por unidade de tempo é adquirido com a execução de um *job*. O termo *Potential Value Density* estende essa definição e inclui no cálculo não somente a tarefa que está sendo considerada, mas também a execução das tarefas das quais ela depende.

2.2.4 *Valor versus Deadline*

Em [17] Buttazzo et al. apresentam um estudo comparativo entre algoritmos de escalonamento com diferentes prioridades e diferentes modos de garantia, a fim de aumentar o desempenho durante situações de sobrecarga. Seguindo as definições apresentadas por Jensen em [2], as tarefas são caracterizadas não apenas segundo seu *deadline*, mas também por um parâmetro *valor*. A análise dos algoritmos ocorre segundo o *valor* acumulado obtido, ou seja, a soma dos *valores* de todas tarefas completadas antes do *deadline*.

É revisto que em condições de sobrecarga não há algoritmo *on-line* ótimo que consiga maximizar o *valor* agregado da execução de um conjunto de tarefas, e, portanto, decisões de escalonamento devem ser feitas. Como exemplo é utilizado o EDF, suscetível ao efeito dominó, o que acarreta uma rápida perda de desempenho durante esse tipo de situação.

Em seus experimentos Buttazzo utiliza *valores* constantes de cada tarefa, através da modelagem de *step functions*, evitando dessa forma o custo computacional adicional necessário ao se usar *valores* dinâmicos durante a execução.

Os objetivos das simulações conduzidas são os seguintes:

- descobrir qual método de atribuição de prioridades atinge o melhor desempenho em condições de sobrecarga;
- entender como e quanto as premissas pessimistas do teste de garantia[7] influem na performance do algoritmo (e quanto um mecanismo de recuperação pode compensar essa degradação).

Um grupo bastante heterogêneo de algoritmos é usado durante os testes, incluindo o tradicional EDF e outros três algoritmos que consideram o parâmetro *valor* durante o processo de definição das prioridades: HVF (*Highest Value First*), onde tarefas com *valor* maior recebem prioridade mais alta, HDF (*Highest Density First*), onde tarefas com densidade maior recebem prioridade mais alta, e MIX (*Mixed Rule*), onde a prioridade é dada como um misto do HDF e EDF, com peso 50% para cada um.

Operando sem nenhuma forma de garantia de execução esses quatro algoritmos são referidos como *plain algorithms*. A falta de cuidado com a carga do sistema os deixa suscetível aos efeitos negativos trazidos pela sobrecarga. Para controlar essas situações de um modo previsível são incluídas duas classes: uma de *guaranteed algorithms* (GEDF, GHVF, GHDF, GMIX), caracterizados por um teste de aceitação, e outra de *robust algorithms* (REDF, RHVF, RHDF, RMIX), caracterizados por um teste de aceitação mais sofisticado e por um mecanismo de recuperação de tarefas.

A classe *guaranteed* executa um teste de aceitação na ativação das tarefas, podendo

[7] *Guarantee test.*

assim detectar a sobrecarga do sistema e rejeitar a nova tarefa. Esse teste evita a sobrecarga, mantendo o sistema sempre abaixo de 100% de utilização. No entanto ele não considera a importância da nova tarefa, que é excluída mesmo possuindo o maior *valor*. Outro problema é que, caso uma tarefa termine a execução antes do tempo previsto, o tempo restante não pode ser usado por tarefas já rejeitadas.

Para resolver esses problemas os integrantes da classe *robust* são capazes de rejeitar tarefas com base no seu *valor*, incluindo também um mecanismo de recuperação de tarefas. Sempre que uma situação de sobrecarga é detectada, a tarefa de menor *valor* é removida do sistema e inserida em uma fila de tarefas rejeitadas. Para executar a recuperação, as tarefas pertencentes à essa fila, ordenadas conforme os respectivos *valores*, são readmitidas no sistema.

Nos testes realizados dois tipos de comparações foram feitas: primeiramente os quatro algoritmos de designação de prioridades foram comparados entre si em três fases, cada fase utilizando uma classe (*plain, guaranteed, robust*) diferente. Em um segundo momento o mesmo escalonador foi comparado entre as três classes definidas.

Da primeira bateria de testes algumas observações importantes são extraídas, dentre as quais se destacam:

- sem esquema de garantia algum, o algoritmo baseado em prioridades mais eficiente em condições de sobrecarga é o baseado na densidade (HDF). Ele exibe uma degradação de desempenho suave e não é muito sensível aos parâmetros das tarefas;
- quando utiliza-se a classe *guaranteed* para rejeitar novas tarefas e evitar sobrecargas, o método de designação de prioridades mais efetivo é o EDF. Problema: demasiadamente pessimista, rejeitando tarefas independentemente do *valor* que possuam;
- considerando-se a classe *robust*, nenhum algoritmo é capaz de ter um desempenho muito melhor que outro em qualquer tipo de carga. No entanto o REDF é o algoritmo mais eficaz na maioria das situações práticas, enquanto o RHDF consegue um *valor* cumulativo maior quando a sobrecarga é realmente alta (acima dos 150%).

Na comparação entre algoritmos semelhantes utilizando diferentes esquemas de garantia o parâmetro escolhido para ser variado foi o tempo efetivo gasto na execução das tarefas. Os resultados apontaram que o teste de aceitação da classe *guaranteed* piora o desempenho de todos algoritmos que consideram *valor* em sua disciplina de ordenação. Por outro lado, a classe *robust* de algoritmos se saiu muito bem em ambos os casos (cargas maiores e menores a 100%), provando que a estratégia de recuperação de tarefas é efetiva para aumentar o *valor* do sistema em situações práticas.

Perante os resultados e observações coletados, conclui-se que a versão dos algoritmos

robust é a mais flexível, por causa do procedimento de recuperação de tarefas. Escalonar por *deadline* e rejeitar por *valor* (como é feito pelo REDF) provou ser a estratégia mais efetiva para uma ampla gama de condições de sobrecarga, embora não seja a melhor solução para todos os casos. Quando há carga menor do que 100%, o EDF é ótimo, e para grandes sobrecargas o RHDF saiu-se um pouco melhor.

Levando-se em conta todas as observações realizadas, Buttazzo sugere que o *valor* acumulado pode ser incrementado se o sistema mudar a estratégia de escalonamento dinamicamente, baseando-se na carga atual.

2.2.5 Flexibilidade Através do *Valor*

McElhone em sua tese de doutorado [3] realiza um extenso estudo sobre a criação de um *framework* capaz de incorporar flexibilidade ao escalonamento de um sistema computacional. Para tanto, vários objetivos secundários são definidos, e os mais importantes são a (1) criação e implementação de um algoritmo de escalonamento, (2) o desenvolvimento de métodos que permitam o acréscimo de serviços adicionais ao processador, sem a necessidade de inclusão de hardware e (3) a demonstração de que o método pode ser implementado em uma linguagem padrão e, portanto, usado na prática sem maiores dificuldades.

Uma das premissas utilizadas para obtenção do objetivo número (1) é de que o escalonador deve utilizar o mesmo processador das tarefas e, logo, deve obter a maior relação custo computacional versus benefício possível. Uma boa alternativa apontada para solucionar esse problema é o uso de TUFs junto com o critério de utilidade acumulada. Porém é notado que a solução *Best Effort* apresentada por Locke [15] apenas sugere o uso de um segundo processador, sem avaliar o *overhead* trazido se isto não for efetuado. Testes realizados com esse objetivo por Tokuda [4] e Wendorf [5] demonstram um *overhead* muito alto, que impede o uso do algoritmo no mesmo processador em que as tarefas são executadas. Outra crítica levantada é que o escalonamento utilizando *Best Effort* apenas aumenta a probabilidade de que as tarefas cumpram os requisitos temporais, e isso é insuficiente quando tarefas tempo real críticas[8], que exigem garantia, estão presentes no sistema.

Fica clara a necessidade de se reduzir a complexidade do algoritmo *Best Effort*, e é debatido se realmente uma função é necessária para a representação do *valor* de uma tarefa. McElhone sugere o uso de uma variável *valor* atrelada à cada tarefa, o que seria algo semelhante a uma *step function*. Por fim um novo modelo é criado, com diferentes "tipos" de tarefas: *obrigatórias, alta utilidade, média utilidade, baixa utilidade e background*, cada um com suas características próprias. Tarefas do tipo *obrigatórias e background* não possuem utilidade; tarefas dos demais tipos possuem

[8] *Hard real-time tasks.*

uma prioridade base, que pode ser aumentada para diferenciar tarefas do mesmo tipo, e com a possibilidade da tarefa mudar de "tipo" (entre os três últimos) dinamicamente ao decorrer da execução.

As seções seguintes da tese visam a modelagem e testes de um modo de aceitação *on-line*, adaptação para uso em sistemas multiprocessados, política de admissão de tarefas e implementação do modelo.

É concluído que o modelo computacional proposto satisfez muitas das demandas exigidas e pode ser implementado, com uma boa relação custo computacional versus benefício. Utilizando-se de algoritmos melhor esforço, aliados ao parâmetro *valor* (porém descartando-se o uso de funções para representá-los), a implementação do modelo em um sistema uniprocessado com uma linguagem padrão, como Ada 95, é dada como viável.

2.2.6 Escolha do Valor

Burns et al. se propõem em [6] a dar uma visão geral do sentido e significado do *valor* em sistemas tempo real, focando o quesito da escolha do *valor* que, segundo os autores, é um tema praticamente ignorado e subestimado em trabalhos anteriores.

A justificativa para o uso do *valor* começa pelo argumento de que nem todos serviços podem ser suportados por um sistema tempo real, passa pelo fato de que os sistemas atuais possuem pouca flexibilidade, principalmente na parte da implementação, e termina com a asserção da necessidade de escalonamento dinâmico, baseada nos seguintes motivos:

- escalonamentos estáticos utilizam recursos de forma ineficiente; como há recursos suficientes para a carga máxima do sistema (tempo de execução no pior caso, tarefas em fase, intervalos entre chegadas no pior caso para tarefas esporádicas etc.) na prática a utilização média é baixa;
- escalonamentos estáticos reagem de forma inflexível a falhas e sobrecargas; embora lidem bem com problemas definidos em seus modelos iniciais, qualquer estado não esperado gera imprevisibilidade e nenhum tipo de garantia.

Neste trabalho é diferenciada *utilidade*, definida como um benefício que o sistema recebe ao completar o serviço, de *valor*, que seria algo aproximado da *utilidade*, porém usado para influenciar o escalonador tempo real. É feita uma separação de preferências entre serviços, ou grupos de serviços, e são enumerados os casos em que uma decisão em tempo de execução (*on-line*) compensa uma decisão em tempo de projeto (*off-line*):

- a decisão é muito mais efetiva se forem considerados dados e/ou condições atuais;
- a decisão conduz a um uso pessimista de recursos se realizada precocemente.

Para facilitar a tomada de decisões em tempo de execução, a relação de preferências

pode ser armazenada e, então, apenas consultada. Todavia essa estratégia é válida apenas se o *overhead* envolvido em sua implantação não for maior ou mais significativo do que os benefícios trazidos por ela. Essa questão é considerada importante, argumentando-se que trabalhos anteriores ignoram o *overhead* em seus testes e, com isso, apresentam resultados equivocados. Vários métodos podem ser utilizados para contornar esse problema, e no atual trabalho é utilizado o parâmetro *valor* atrelado a serviços e subserviços para desempenhar essa função.

A descrição do chamado Escalonamento Baseado em Valor[9] é simples: trata-se de um problema de decisão envolvendo a escolha de uma coleção de serviços a serem executados que tragam o "melhor" resultado (em um determinado momento, há vários serviços prontos para serem executados e recursos insuficientes para atendê-los).

As seções seguintes são usadas na formulação e provas matemáticas do problema e em testes para resolução do mesmo. É criado um *framework* que considera serviços em diferentes estados, e usa esses estados no processo de escolha de quais serão executados, assim como "modos" de execução de um mesmo serviço. Fatos que devem ser destacados são o uso de *valores* constantes, a relação ordinal entre os grupos de alternativas de execuções e a relação cardinal entre as alternativas dentro de um mesmo grupo.

Há uma seção especial acerca do processo de escolha, sob o argumento de que sem *valores* significativos a teoria de escalonamento baseado em *valor* perde completamente o sentido. São listados então os dois problemas básicos encontrados no processo de escolha de *valor*:

Representation Problem: saber se existe uma função que represente as preferências na escolha das alternativas;

Construction Problem: saber **como** construir tal função na prática.

Outro fato importante é que não é necessário ter uma função exata; preferências são o ponto chave do processo, e a função é apenas um método útil de representar essas preferências. Após a apresentação de um exemplo prático, é reiterada a tentativa deste trabalho em transformar toda uma série de estudos teóricos em algo prático e utilizável. Comenta-se que há espaço para muitas pesquisas futuras, especificamente na área de escolha do *valor*.

2.2.7 Exemplos do Uso de *Valor*

Davis et al. em [18] apresentam o uso de *valor* em um sistema para controle autônomo de veículos. Neste sistema, qualquer falha, seja ela de *software*, *hardware* ou sensor, pode ser catastrófica e, portanto, precisa ser previsível e tratável. Além

[9] *Value-Based Scheduling*.

disso, o sistema deve ser capaz de apresentar uma degradação suave no desempenho em situações de sobrecarga. Os objetivos potencialmente conflitantes traçados para esse estudo são: (1) garantir inicialmente os serviços críticos relacionados à segurança e funcionalidade para que resultados minimamente aceitáveis sejam apresentados sempre e (2) maximizar a utilidade do sistema, determinada pela freqüência, correção temporal[10], precisão e segurança dos resultados.

Todas as rotinas pertinentes a um sistema de controle autônomo de veículos são descritas e classificadas em dois grandes grupos: *obrigatório* e *opcional*. Tarefas relacionadas à segurança e funcionalidade mínima do sistema são inseridas no grupo *obrigatório*, e são consideradas mais prioritárias do que as tarefas presentes no grupo *opcional*. Esse modo é o sugerido por Davis para atender ao objetivo (1), pois o tratamento desses dois grupos é distinto: enquanto as tarefas do grupo *obrigatório* são executadas sempre, sem exceções, as do grupo *opcional* passam por um processo de admissão e mais tarde são escalonadas através do método *Slack Stealing* [19].

O segundo objetivo do trabalho se resume a encontrar uma maneira de executar o máximo de tarefas opcionais, ou seja, maximizar a utilidade do sistema. Nesse processo são utilizadas as *Time/Utility Functions* e o resultado final é avaliado pela utilidade acumulada, ou *Utility Accrued* (UA). Um novo método de aceitação é desenvolvido, intitulado *Adaptive Value Density Threshold* (AVDT), baseado no método *Best Effort* proposto por Locke em [15]. A grande diferença presente no algoritmo AVDT é a existência de um filtro que pré-seleciona tarefas utilizando um limite mínimo para corte, esperando dessa forma superar a principal falha do método base: o alto *overhead* do algoritmo.

Davis atesta que o número de tarefas executadas do grupo *opcional* influi muito na utilidade final obtida pelo sistema como um todo. Os testes empregando *valor* e a nova política de admissão mostraram-se uma alternativa viável para esse tipo de sistema. A idéia de um limiar mínimo para a aceitação de tarefas pode ser estendida para o escalonamento em uma versão futura.

Em [20] Bondavalli trata da melhor forma de uso de recursos por sistemas tempo real orientados a objetos, e introduz o conceito de *valor* como parâmetros desses objetos. Sua proposta é a criação de um componente chamado *planner*, que implementa a política de admissão/rejeição baseado no *valor* dos objetos e almeja maximizar o uso dos recursos, isto é, maximizar a utilidade agregada obtida. É concluído que o uso do componente criado consegue introduzir flexibilidade ao sistema a um custo computacional muito baixo.

Wu em [21] apresenta um novo algoritmo de escalonamento para sistemas embutidos movidos a bateria. Dentre os objetivos fundamentais estão a maximização da

[10] *Timeliness.*

utilidade do sistema e da eficiência da utilização da energia. A solução introduzida escalona tarefas em um tempo polinomial e apresenta características importantes, como otimalidade em caso de *underload* e comportamento temporal assegurado estatisticamente. Como pode-se prever, o modelo TUF/UA é empregado para a obtenção da maior utilidade possível do sistema com o menor consumo de energia.

2.2.8 Estado da Arte

Desde o ano de 2003 Jensen vem trabalhando com TUF/UA em conjunto com a Universidade de *Virginia Tech*. Após vários trabalhos e artigos publicados especificamente nesta área é feito um resumo dos avanços obtidos, apresentados em [22] por Ravindran et al.

Neste trabalho é alegado que a base da prática em sistemas tempo real - o artefato prioridade - e o estado da arte atual na teoria de sistemas tempo real - otimalidade baseada em *deadlines* - são completamente inadequados para especificar objetivos, argumentar sobre comportamento temporal e gerenciar recursos que podem satisfazer de forma fiel requisitos em sistemas dinâmicos.

É defendido que as TUFs e o critério de otimalidade UA representam um método mais genérico, flexível e adaptativo, e que os últimos avanços nessa área trazem significativas melhorias a problemas conhecidos. Um apanhado geral nessas melhorias é feito.

A maior parte das restrições temporais de aplicações são expressas e tratadas através do parâmetro prioridade. Porém prioridades possuem significantes desvantagens, abaixo descritas:

- geralmente não é tratável mapear restrições temporais para prioridades, e essa prática resulta em perda de informações;
- prioridades não conseguem expressar urgência, pois seria necessário conhecimento global de todas prioridades - o que é geralmente difícil de se conseguir durante o desenvolvimento de sistemas;
- a urgência de uma tarefa é tipicamente ortogonal à importância relativa da tarefa, porém uma prioridade não consegue expressar ambos.

A teoria tradicional de tempo real visa superar essas desvantagens provendo modelos de aplicações com a abstração direta de restrições temporais, e usando essa abstração para o gerenciamento de recursos. No entanto, essa teoria é fundamentalmente limitada à restrição de tempo *deadline*. Escalonamentos baseados em *deadlines* possuem as seguintes desvantagens:

- um *deadline* é ou (i) um valor binário no sentido de ser ou não cumprido ou (ii) uma expressão linear pela penalidade ao atraso. Portanto *deadlines* também não

conseguem distinguir entre urgência e importância;

- escalonamentos clássicos baseados em *deadlines* sofrem do indesejado efeito dominó durante situações de sobrecarga (vide EDF);
- *deadlines* não conseguem expressar restrições temporais que não são binárias nem lineares, no sentido de que a utilidade da tarefa varia conforme o instante em que for finalizada.

É necessário notar que o paradigma TUF/UA supera essas desvantagens. Algoritmos baseados em UA conseguem cumprir todos *deadlines* em situações normais, e quando ocorrem sobrecargas favorecem atividades mais importantes em detrimento das mais urgentes.

Apesar da generalidade e superioridade do escalonamento baseado em TUFs e UA, esses métodos também possuem defeitos que aparentemente os impedem de serem adotados em larga escala. Os mais significativos são:

- falta de garantia geral na correção temporal dos sistemas TUF/UA - exceto no caso especial de otimalidade de TUFs com *step functions* em condições de *underload*;
- falta de ferramentas de suporte para criar TUFs e realizar análises de UA;
- falta de algoritmos UA que considerem qualidade de serviço de sistemas embutidos (nesse tipo de sistema há outros parâmetros cruciais, como consumo de energia e gerenciamento de memória);
- algoritmos UA adicionam um *overhead* maior do que algoritmos baseados em prioridades/*deadlines*.

Durante o restante do trabalho é mostrado resumo geral dos avanços práticos obtidos nos últimos anos em diferentes áreas onde TUFs estão sendo usadas, como, por exemplo, escalonamento eficiente de energia, escalonamento com compartilhamento de recursos e escalonamento eficiente de memória.

Conclui-se que o paradigma TUF/UA é importante em sistemas dinâmicos tempo real, e que o estado da arte avançou de forma significativa recentemente. Novas variedades desenvolvidas superam desvantagens dos métodos originais e expandem a área de cobertura do escalonamento TUF/UA. Por fim, acredita-se que sistemas tempo real emergentes possam se beneficiar substancialmente do paradigma TUF/UA, o que permitirá alavancar a adaptabilidade e flexibilidade do paradigma, sem deixar de usufruir das vantagens da teoria tradicional (otimalidade na correção temporal sob *underload*, garantia estatística da correção temporal).

2.3 Política de escalonamento TAFT

Esta seção apresenta um resumo dos trabalhos envolvendo a política de escalonamento *Time-Aware Fault-Tolerant* (TAFT). Inicialmente é mostrado o modelo de *TaskPairs* no qual o TAFT é baseado; logo após o conceito inicial e primeiros resultados práticos são exibidos; a seguir a nova técnica de escalonamento sugerida é descrita; por fim, é visto seu uso nas diferentes áreas, além de extensões implementadas.

2.3.1 Escalonamento Segundo o Modelo de ***TaskPairs***

O escalonamento baseado em par de tarefas (*TaskPairs*) é apresentado em [23] por Streich, com a intenção de unir os conceitos de garantia (de uma atividade) e de tratamento de exceção (devido a um *timeout*). A proposta usa um escalonador *on-line* e envolve a construção de uma tarefa como um bloco *try-except*, dividindo-a em um par organizado segundo uma parte principal (*MainTask*) e uma parte de exceção (*ExceptTask*).

O problema típico de sistemas tempo real - a obtenção de um comportamento temporal determinista - pode ser resolvido através de uma análise *off-line* completa da aplicação e de todo ambiente envolvido. Para aplicações tempo real críticas, onde todas as tarefas devem ser executadas e todas as restrições temporais devem ser cumpridas, não há muitas alternativas, e o WCET é usado para garantir essa correção temporal. A subutilização dos recursos e uma estruturação estática do sistema inteiro são apenas alguns dos problemas relacionados ao uso de WCETs, visto que essa metodologia pode se mostrar falha em casos onde:

- a carga do sistema pode variar dinamicamente;
- os tempos de execução no pior caso não podem ser calculados ou medidos;
- mudanças no ambiente são inerentes da aplicação e, portanto, esperadas.

Tal conjunto de problemas levou à criação de sistemas tempo real dinâmicos como, por exemplo, o *kernel* Spring [24] e mais tarde o escalonador *Robust Earliest Deadline* (RED) [25]. Sistemas desse tipo têm em comum a característica de escalonar tarefas de maneira *on-line*, cuja principal vantagem é que o estado atual do sistema pode ser considerado perante uma decisão.

Contudo, se nenhuma restrição para criação de tarefas for imposta e todas novas tarefas forem incluídas automaticamente na fila de execução, violações de *deadlines* podem ocorrer, e torna-se necessária uma política para tratá-las.

A predição *on-line* mais severa que pode ser feita baseia-se no conceito de *garantia* de uma tarefa, ou seja, aceitá-la apenas se a mesma puder ser completada; em caso contrário a tarefa não chega a entrar na fila do escalonamento. Esse método é adequado a várias aplicações, especialmente aquelas onde erros temporais levam a estados que

não podem ser tratados ou recuperados, porém mostra-se bastante limitado e exige o conhecimento prévio de todos WCETs. A política apresentada permite que tarefas "tentem" executar, mesmo não garantidas, e, caso ocorra uma falha, o sistema fique em estado operacional (sem inconsistências).

Um método tradicional de realizar o tratamento perante uma perda de *deadline* recai sobre a construção *try-except*, ilustrada pelo trecho de código 2.2.

```
try within deadline {
    ...
}
except
    abort
end
```

Figura 2.2: Modelo de bloco *try-except* [23]

O problema é que, sem restrições temporais no código, no momento em que a falha ocorre a reação terá que ser executada mesmo não havendo tempo para isso (o *deadline* já foi perdido).

O modelo de escalonamento baseado em *TaskPairs*, *TaskPair-Scheduling Model* (TPS), evita que esse problema ocorra, tratando a exceção como uma tarefa separada com tempo de execução próprio. O TPS pode ser visto como uma junção do método de garantia de execução de tarefas com as estratégias de tratamento de exceções encontradas em várias linguagens.

A idéia é escalonar a *MainTask* se e somente se o escalonador garantir que seu término, ou o término da *ExceptTask*, ocorrerá antes do *deadline* único definido pelo *TaskPair*. A diferença desse método de garantia para outros métodos, como o encontrado no *Spring kernel*, por exemplo, é a divisão da expressão *try-except* em unidades escalonáveis separadas. E, em contraste com outros mecanismos de tratamento de erros que são executados após a violação do *deadline*, o TPS executa o código de exceção de forma preventiva. Com construção similar a do bloco *try-except*, um *TaskPair* é executado segundo o código da figura 2.3.

O método de escalonamento dos *TaskPairs* utilizado é o seguinte: *MainTasks* seguem a estratégia *Round-Robin*, todas com prioridade baixa; *ExceptTask* são escalonadas com prioridade alta, de um modo que sejam executadas o mais tardar possível[11]. A execução das *ExceptTasks* ocorre em instantes de tempo pré-definidos (*time triggered*) e é atômica, ou seja, não permite preempções. Assim que um *TaskPair* é finalizado seus recursos reservados e não utilizados são liberados para outros *TaskPairs*.

[11]Neste trabalho não é definido um método formal de escalonamento para *ExceptTasks*.

```
if (guarantee (TP, deadline)) {
    try {
        <MainTask>
    except
        <ExceptTask>
    end
}
else  // no guarantee
```

Figura 2.3: Modelo de execução de um *TaskPair* [23]

MainTasks também podem estar sujeitas a um teste de garantia e, no caso de serem aceitas, as *ExceptTasks* correspondentes tornam-se desnecessárias. No entanto garantia de *TaskPairs* leva vantagem em cima da garantia de *MainTasks* em dois casos:

- o WCET da *MainTask* é muito superior ao seu tempo médio de execução;
- o WCET da *MainTask* é muito superior ao WCET da *ExceptTask*.

Além disso, a principal vantagem da garantia de *TaskPairs* é que eles podem ser escalonados mesmo que a *MainTask* tenha um WCET desconhecido, ou seja, apresente um comportamento anômalo e não monitorável. É então introduzido o chamado "tempo de execução otimista" (*Optimistic Case Execution Time* - OCET), que nada mais é do que uma medida rústica do tempo médio de execução de uma tarefa. Essa medida deve ser longa o suficiente para a tarefa tenha chances razoáveis de execução, porém mais curta do que o WCET (que pode ser conhecido ou não).

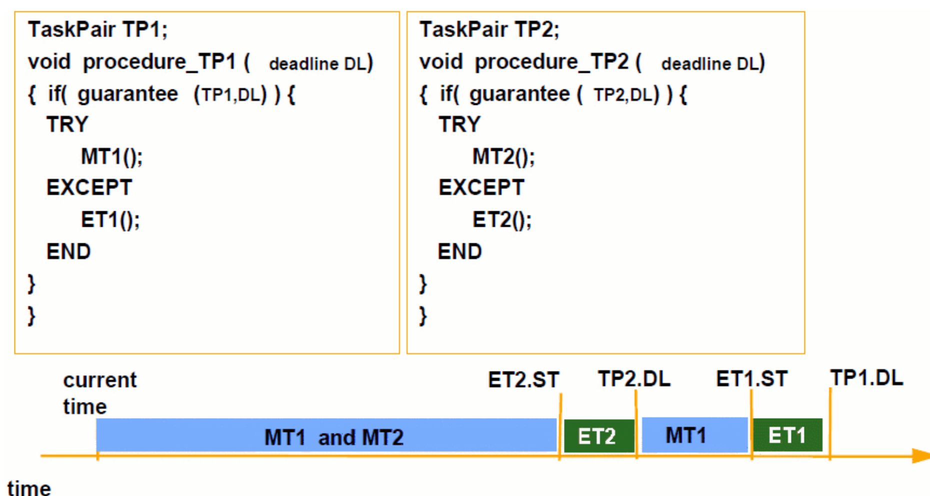


Figura 2.4: Exemplo de escalonamento de *TaskPairs* [26]

O escalonador utiliza então o OCET para o teste de garantia de execução do *TaskPair*, que se abrevia à verificação da existência de tempo disponível suficiente para a

execução da *MainTask*, segundo seu OCET, e da *ExceptTask*, segundo seu WCET. Ou seja: se há uma chance razoável, deve-se tentar executar a tarefa. A figura 2.4 ilustra um escalonamento envolvendo dois *TaskPairs*, onde as abreviaturas DL e ST significam respectivamente *deadline* e tempo de início (*start time*).

Nett e Streich em [26] apresentam as experiências e conceitos de escalonamento utilizados na implementação de um software de controle de um robô construído para inspeções de tubos. Os principais requisitos da aplicação são comportamento dinâmico durante a execução e alta disponibilidade em situações de falha (ou controle anti-falhas), e não deixam dúvidas de que o escalonamento utilizado deve ser realizado em tempo de execução.

Algumas propostas existentes de escalonamento são discutidas, porém todas trabalham com o tempo de execução no pior caso, cujos problemas já foram citados: WCETs são difíceis de serem previstos, e pode existir uma variação muito grande entre o WCET e o tempo médio de execução. Dessa forma, trabalhar com WCETs requer uma abordagem de prevenção de falhas, com métodos de isolamento (para prevenir o efeito dominó) e tratamento de erros temporais[12] (*timing faults*) que possam ocorrer.

O método sugerido para a tolerância de erros temporais é o uso de *TaskPairs*. O fato de uma *MainTask* ser considerada crítica ou não é controlado pelo tempo reservado para sua execução. Não reservar tempo algum indica uma estratégia puramente de melhor esforço para a execução da tarefa, enquanto que a reserva de um tempo equivalente ao WCET inutiliza o uso da *ExceptTask* correspondente. Possíveis soluções intermediárias apresentadas sugerem estratégias de reserva de tempos mínimo, médio, esperado ou otimista para a execução da *MainTask*.

A adaptação para o software de controle transformou as operações em tarefas, e conseqüentemente em *TaskPairs*. Nett e Streich afirmam que nesse software a abordagem se mostrou uma excelente base para a construção de sistemas tempo real altamente flexíveis. Trabalhos futuros prevêem a continuação do uso do modelo de *TaskPairs* para a implementação das tarefas.

2.3.2 Conceito TAFT de Escalonamento

Nett et al. alegam em [27] que o conceito tradicional de sistemas tempo real, geralmente ilustrados como dispositivos embutidos isolados, está ultrapassado. Sistemas tempo real estão ficando maiores, mais complexos, sendo executados de forma fisicamente distribuída e terão que ser capazes de funcionar em ambientes incertos.

A proposta é integrar sistemas tempo real com sistemas tradicionais em um único sistema heterogêneo como uma solução de *middleware*. O conceito de orientação a

[12] Um erro temporal ocorre quando o tempo de execução de uma tarefa difere do que foi especificado previamente.

objeto, implantado no ambiente *middleware* com sucesso, não pode ser simplesmente adaptado para sistemas tempo real: operações básicas dessa teoria, como uma simples troca de objetos que implementam a mesma interface, pode tornar o escalonamento não factível devido às mudanças dos requisitos do sistema.

A propriedade de invocar objetos não tempo real através de um domínio tempo real e ainda assim manter um grau de previsibilidade na aplicação tempo real é intitulada *time-awareness*. Essa propriedade pode ser vista como a solução geral da questão encontrada: "Como adicionar previsibilidade em um sistema imprevisível?".

Para resolver esse problema o sistema deve ser capaz de lidar com certo nível de incerteza dos componentes e ser flexível o suficiente para adaptar-se às pequenas mudanças em seu ambiente, não previstas durante a fase de projeto.

O primeiro empecilho encontrado durante o escalonamento é tentar manter consistência e previsibilidade em um sistema tempo real onde tarefas são aceitas pelos seus tempos esperados de execução, ou seja: tarefas cujos WCETs são desconhecidos ou, em uma situação mais pessimista, podem mudar durante a execução do sistema. Abordagens tradicionais são classificadas como insuficientes, pois recaem no uso do WCET e exigem conhecimento prévio de vários parâmetros temporais. A definição desses outros parâmetros temporais (como tempo de chegada, por exemplo) consiste no segundo empecilho encontrado. É reivindicado que um monitoramento do sistema de forma ampla e contínua pode sanar essa dificuldade. Esse monitoramento seria realizado no nível de *middleware*, deixando-o independente de aplicação e adequado para permitir à uma maior gama de programas o comportamento *time-aware*.

A solução apresentada reside no uso de uma estratégia que engloba dois componentes principais: um sistema de monitoramento durante a execução e um escalonador *on-line* tolerante a falhas. É introduzido então o sistema de escalonamento *Time-Aware Fault-Tolerant* (TAFT), formado pelo escalonador FT tolerante a falhas[13], que impõe previsibilidade ao sistema, e pelos componentes de monitoramento, responsáveis pelo conhecimento sobre o comportamento da aplicação e que servem como base para a adaptabilidade.

Monitoramento

O monitoramento tem como obrigação coletar todo tipo de parâmetros temporais. Os dados são filtrados, analisados e explorados para derivar o tempo esperado de execução (ECET), que serve como entrada no algoritmo de escalonamento. De grosso modo, o ECET representa uma medida de tempo que a tarefa necessita na maioria dos casos para ser finalizada. Formalmente falando e sendo t uma instância da tarefa T, o ECET pode ser definido da seguinte maneira:

[13] *Fault-Tolerant.*

$\mathbf{ECET}_{t,\alpha}$ = tempo de CPU necessário pela instância t para haver uma probabilidade α de a tarefa ser completada dentro desse tempo.

Ao contrário do WCET, que é estático, o ECET pode variar dependendo do código, do estado do sistema e dos parâmetros da tarefa. Isso significa que o ECET de uma tarefa pode mudar entre uma execução e outra, e o escalonador precisa estar preparado para essa mudança. No entanto é apresentada uma propriedade de "localidade" bastante alta, e apenas mudanças marginais são observadas entre execuções conseguintes.

Escalonador FT

O uso de ECETs permite que apenas um percentual das execuções ocorra dentro do tempo delimitado; é responsabilidade do escalonador FT dar conta das execuções restantes e garantir que o sistema permaneça em um estado consistente. Esse tratamento de parâmetros temporais incertos, ou possivelmente errôneos (devido à sua natureza), é realizado através da troca do funcionamento ótimo pela correção temporal.

O mecanismo de tolerância utilizado para negociar falhas temporais, de forma que *deadlines* nunca sejam perdidos, é a construção de tarefas segundo o modelo de *TaskPairs*. Cada *TaskPair* (TP) é constituído por uma *MainPart* (MP) e uma *ExceptPart*[14] (EP). A funcionalidade encontrada em uma EP deve ser mínima, apenas assegurando que:

- a aplicação controlada fique em um estado seguro;
- o sistema de controle esteja em um estado consistente.

O *deadline* da EP é idêntico ao do TP, e seu término é garantido pelo escalonador que reserva explicitamente os recursos exigidos (ou seja, o WCET). Já o tempo limite de execução (ou *deadline* interno) de uma MP é calculado como o *deadline* do TP menos o WCET da EP. O teste de aceitação de um TP ocorre de maneira simples: um resultado positivo é retornado se o ECET da MP e o WCET da EP podem ser escalonados antes do *deadline* do TP. É preciso lembrar que isso depende muito do ECET da MP, que é monitorado e estimado pelo próprio sistema. O WCET da EP deve sempre ser conhecido, porém essa não é uma restrição muito severa, pois assume-se ser um tempo no pior caso relativamente curto se comparado ao tempo médio de execução da MP. Considerando-se:

$C_i = \mathbf{ECET(MP}_i\mathbf{)}$ tempo de execução estimado da MP da tarefa i;

$E_i = \mathbf{WCET(EP}_i\mathbf{)}$ pior caso de execução da EP da tarefa i.

podemos representar essa premissa por $C_i \gg E_i$.

[14] São os termos equivalentes para *MainTask* e *ExceptTask*.

Tem-se ainda que na maioria dos casos ECETs são consideravelmente menores que WCETs, gerando normalmente uma soma dos tempos C_i e E_i muito menor que o tempo que seria obtido caso uma estimativa de pior caso fosse determinada para a MP. Resumidamente: $C_i + E_i \ll \mathrm{WCET}(\mathrm{MP}_i)$. Desta forma, mesmo estimativas bastante pessimistas de E_i não implicam em utilização de recursos comparáveis com a MP. Como conseqüência direta, chega-se a escalonamentos factíveis em casos onde um escalonamento tradicional baseado em WCETs não encontraria solução.

O comportamento do escalonador é descrito de modo a evitar a violação de *deadlines* de *TaskPairs* em situações normais. É então estudada uma estratégia para situações de sobrecarga, onde não existe como cumprir todos os requisitos do sistema. A proposta é escalonar tarefas por "importância", de modo que o cancelamento de execuções seria feito pela ordem inversa de importância das tarefas. No modelo TAFT a importância pode ser representada pelo parâmetro α, presente na definição de $\mathrm{ECET}_{t,\alpha}$ da página 22. A premissa é que quanto mais próximo de 1 for α, mais próximo do WCET estará o ECET, mais recursos serão reservados para a tarefa e maiores serão as chances dela ser executada.

O restante do trabalho trata de detalhes utilizados na implementação da política TAFT, e as conclusões finais salientam que a principal diferença para outros métodos de tolerância de falhas é a abordagem de tratamento de exceções, alocada pelo TAFT como uma tarefa normal deixada sob controle do escalonador tempo real. Outro avanço importante é a possibilidade de escalonamento de tarefas cujos tempos de execução não são exatos (ou baseados no pior caso). Pelo simples fato de ECETs serem mais próximos da realidade, um número mais elevado de tarefas é aceito e um rendimento maior aparece, resultado da melhor utilização de recursos.

2.3.3 Determinação dos Tempos de Execução das *MainParts*

Em sua tese de doutorado [28] Gergeleit define um componente de monitoração utilizado para coletar os tempos de execução dos TPs e gerar estimativas a respeito do ECET das MPs. Cada MP é associada a um parâmetro $C_{j,\alpha}$, que descreve o tempo de CPU que deve ser atribuído à MP do TP τ_j de modo a obter uma probabilidade α que τ_j finalize sem executar sua EP.[15]

O valor $C_{j,\alpha}$ pode ser derivado de uma função de distribuição de probabilidade[16] (PDF), ilustrada na figura 2.5.

Os símbolos em forma de $\times$ do gráfico denotam os valores da PDF, e a área sombreada sob os pontos representa as somas das probabilidades até aquele tempo de execução (i.e. a distribuição de probabilidade). Esta soma excede a probabilidade α para o C

[15]O parâmetro α de $C_{j,\alpha}$ é semanticamente equivalente à definição homônima em $\mathrm{ECET}_{t,\alpha}$ (p. 22).
[16]Probability Distribution Function

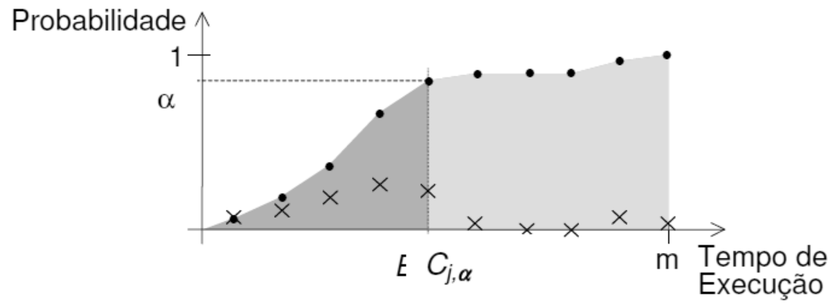


Figura 2.5: Curva da PDF para estimação do parâmetro C e soma das probabilidades em determinado instante [29]

requisitado no eixo do tempo de execução (a área sombreada escura). Para ser mais preciso, α é o quantil (ou separatriz) da PDF. A proposta de Gergeleit também defende que ao assinalarem-se diferentes valores de α para diferentes tarefas é possível expressar uma noção de importância para cada tarefa.

Segundo a estratégia de escalonamento prevista, o $C_{j,\alpha}$ de uma tarefa deve ser associado com a sua importância, de forma que quanto maior for o valor α maior é a importância relativa da tarefa.

2.3.4 Mecanismo de Escalonamento do TAFT

Em sua tese de doutorado [29] Becker apresenta formalmente um mecanismo de escalonamento para a política TAFT.

Uma aplicação TAFT pode ser definida como sendo um conjunto $\prod$, constituído por τ_i tarefas independentes entre si e projetadas como TPs. Cada TP_i é uma tupla, composta por uma *MainPart* MP_i e uma *ExceptionPart*[17] EP_i, com *deadline* D_i e com tempo de ativação T_i. Este último pode ser interpretado como período nos TPs periódicos, ou como intervalo mínimo de chegada para TPs esporádicos. Em suma, descreve-se uma aplicação TAFT composta por n TPs como:

$$\prod = \{\tau_i = (T_i, D_i, C_i, E_i), i = 1 \text{ até } n\}$$

Uma EP deverá ser executada se e somente se a MP correspondente não terminar até o instante de tempo mais tardio em que a EP deve ser disparada para que seu término não viole o limite D_i. Conseqüentemente, a execução de uma EP implica em abortar a MP respectiva. A figura 2.6 exibe os parâmetros de um TP.

Claramente pode-se perceber duas estratégias distintas para o escalonamento das MPs e EPs, o que leva a um escalonamento dito hierárquico: criar um escalonamento para as EPs implica em determinar o seu instante de ativação mais tardio (denotado por

[17]Embora a nomenclatura esteja diferente, referem-se respectivamente às *MainTask* e *ExceptTask* de Streich [23] e Nett e Streich [26], e às *MainPart* e *ExceptPart* de Nett et al. [27].

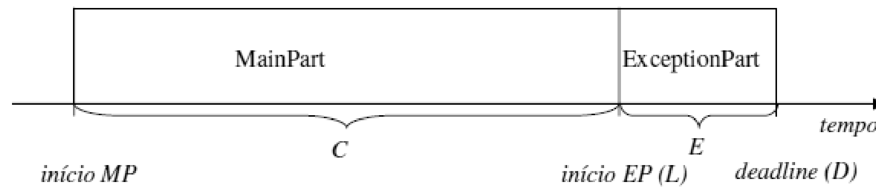


Figura 2.6: Estrutura de um *TaskPair* segundo o TAFT [29]

L), sendo que o instante L é usado como *deadline* para a respectiva MP. Em relação ao escalonamento das MPs, devem ser utilizadas estratégias que maximizem a utilização do processador e que mantenham um alto nível de execuções bem-sucedidas. Assim, podem ser usados algoritmos de escalonamento com estratégia de melhor esforço, a fim de otimizar o uso do sistema (porém sem oferecer garantias de execução), ou então algoritmos de escalonamento tempo real dinâmicos, que mantêm uma alta utilização do processador com a vantagem de contar com garantias de execução.

Mecanismo de Escalonamento Proposto

É proposta uma estratégia de escalonamento de dois níveis, com primeiro nível escalonando EPs e o segundo MPs. De acordo com esta proposta, a saída produzida pelo primeiro nível sempre terá uma maior prioridade do que a saída produzida pelo segundo nível, pois no pior caso, quando nenhuma MP conseguir terminar antes do *deadline*, todas EPs precisam ser executadas. Não são feitas restrições relativas ao instante de ativação de MPs. Por outro lado, as EPs possuem uma restrição severa em relação ao seu instante de ativação L_i que, conforme já mencionado, deve ocorrer o mais tarde possível, para maximizar o tempo disponível para as MPs, porém cedo o suficiente para garantir sua finalização antes do *deadline* D_i. A formalização da estratégia proposta é feita na definição 2.3.1:

Definição 2.3.1 *O escalonador proposto é um escalonador de dois níveis, que funciona para um conjunto tarefas periódicas* $\prod = \{\tau_i = (T_i, D_i, C_i, E_i), i = 1 \text{ até } n\}$ *projetadas como TPs, sendo que o* deadline D_i *é igual ao período* T_i*, o tempo de execução* C_i *da MP é caracterizado como ECET e o tempo de execução* E_i *da EP é caracterizado como WCET.*

O segundo nível, encarregado de escalonar as MPs, usa o algoritmo EDF, devido à sua otimalidade. Adicionalmente assegura-se um maior número possível de MPs executadas, pois o EDF maximiza a utilização da CPU devido à característica de escalonar as tarefas o mais cedo possível.

Considerou-se neste trabalho o uso do algoritmo *Latest Release Time* (LRT) [11], ou *Earliest Deadline as Late as possible* (EDL) [9], para escalonar as EPs, representando o

primeiro nível de escalonamento. Este algoritmo pode ser interpretado como um EDF reverso, tratando tempo de ativação como *deadline* e vice-versa. O LRT é provado ser ótimo (vide [9]) dentro das mesmas condições que o EDF é ótimo: é possível escalonar um conjunto de EPs periódicas independentes entre si, preemptivas e com *deadline* igual ao período, sempre que o fator de utilização do conjunto for menor ou igual a 1. Portanto, o escalonamento das EPs é "pessimista" no sentido de que assume a condição de pior caso, em que todas as EPs precisam ser executadas além das MPs. Este critério é fundamental para a elaboração do teste de aceitação.

Aplicação Prática do TAFT

A fim de validar o mecanismo de escalonamento desenvolvido incorporou-se a proposta de escalonamento TAFT ao sistema operacional tempo real RTLinux [30]. A escolha do RTLinux como plataforma alvo se deu pelo fato do mesmo possuir um mecanismo de despacho bem estruturado e estritamente baseado em prioridades e também devido ao seu código fonte aberto, o que permite a realização de modificações direto no código do escalonador.

O detalhamento dos testes e a aplicação escolhida são descritos em [7], onde Becker et al. dissertam sobre os desafios encontrados no processo de cooperação entre vários dispositivos robóticos. O exemplo dado é de futebol de robôs, no qual os vários "jogadores" devem atuar de forma coordenada apresentando o comportamento de um time com um objetivo único. As dificuldades aumentam à medida que o ambiente encontrado é desconhecido, imprevisível e dinâmico, e os dispositivos móveis possuem restrições de espaço e energia.

Encargos como a movimentação dos robôs e o reconhecimento de objetos devem utilizar dados compartilhados, e a interação com o ambiente físico resulta na necessidade de tarefas tempo real críticas e um escalonador altamente eficiente dos escassos recursos (principalmente CPU e largura da banda de comunicação).

É observado que a política TAFT, através da divisão de tarefas em MP e EP, propicia a ocorrência de falhas por omissão (na impossibilidade da execução de uma MP) ao invés de falhas temporais[18] (sem abortar tarefas) ou falhas de valor arbitrário[19] (abortando a tarefa e sem limpeza dos dados). Tais falhas por omissão podem ser toleradas através de várias redundâncias funcionais, geralmente específicas da aplicação.

Uma alternativa descrita para exploração de redundâncias é baseada na estruturação da MP como um algoritmo *any-time* [31], onde resultados precoces são produzidos rapidamente e melhorados o máximo possível durante o tempo de execução definido. Essa estratégia é utilizada na aplicação de fusão de sensores, e cabe então

[18] *Timing failures.*
[19] *Arbitrary value failures.*

à EP, quando executada, o dever de retornar o resultado produzido até o momento da interrupção da MP. Desse modo é explorada a redundância funcional para tolerar falhas causadas pela estimativa errada de tempos de execução.

O protótipo implementado no *RTLinux* tem o comportamento do algoritmo, especialmente em condições de sobrecarga, analisado sob uma variação do *Benchmark Hartstone* [32]. As conclusões atestam a eficiência do TAFT, capaz de sustentar a CPU com índices elevados de utilização sem perder os *deadlines* dos TPs. Em situações de sobrecarga o ambiente se mostra bastante estável, conseguindo manter um bom índice de tarefas finalizadas sem perdas de funcionalidade, além de se mostrar imune aos problemas causados pelo efeito dominó.

Outra conclusão importante está no fato do parâmetro α-quantil não ter satisfeito seu propósito de representar a importância do TP, conforme prática sugerida em Nett et al. [27] e Gergeleit [28]. Conseqüentemente, o autor faz uma ressalva quanto a seu uso, a constar:

> [...] através dos experimentos foi possível constatar que o parâmetro α-quantil, utilizado para expressar o grau de precisão relacionado com o tempo de execução, não se apresenta como uma boa alternativa para expressar a importância das tarefas. Testes adicionais realizados mostram que algoritmos "baseados em *valores*", como o HDF, apresentam-se como uma boa maneira de solucionar esta carência.

2.3.5 Extensões do Conceito TAFT

Schemmer em sua tese de doutorado [1] desenvolve um *middleware* para sistemas embutidos móveis cooperativos atuando em um ambiente comum com um meio *wireless* de comunicação. Um capítulo especial é destinado ao escalonamento local dos recursos do processador, onde analisa-se o problema encontrado quando o tempo de execução é dependente do ambiente, uma característica presente em tarefas que exercem o tratamento de informações adquiridas externamente (através de um sensor, por exemplo).

A solução encontrada para o escalonamento deste tipo de tarefa é o emprego do conceito TAFT. Através de seu uso, correção temporal, tratamento de erros e isolamento de falhas se tornam características garantidas pelo escalonador. No entanto a estratégia TAFT de escalonamento não se adapta 100% ao modelo de tarefas proposto para o domínio da aplicação, e algumas extensões precisam ser desenvolvidas:

→ para escalonar demandas de processador provenientes dos protocolos de comunicação do *middleware* o modelo deve aceitar, além das tarefas periódicas, tarefas aperiódicas;

→ para permitir que o resultado produzido por um *TaskPair* seja usado como dado de entrada em outro, o modelo deve permitir restrições de precedência entre os *TaskPairs*.

Schemmer também acrescentou ao novo modelo de escalonamento, além das duas extensões citadas, a possibilidade de explorar redundâncias inerentes da aplicação descrita, como redundâncias funcionais (MP não precisa ser executada até o fim para fornecer resultados suficientes), espaciais (a mesma parte de um ambiente é observada por vários sistemas móveis, logo um pode suprir informações faltantes de outro) e temporais (que diz respeito ao número de tarefas que devem retornar resultados, pois nem sempre a resposta de todas é necessária).

Por fim, é concluído que, embora o conceito original do TAFT não seja adequado para a aplicação alvo, o modelo estendido adapta-se muito bem aos objetivos traçados. Experimentos realizados mostram que usado em conjunto com métodos para explorar as redundâncias da aplicação o protótipo garante uma execução segura, mesmo sem limites conhecidos de WCET das tarefas.

2.4 Resumo

Este capítulo apresentou duas teorias completamente distintas que visam um objetivo geral bastante similar: superar as deficiências do modelo de escalonamento tradicional de sistemas tempo real, fundamentado em cima do *deadline*.

De um lado temos a teoria do escalonamento baseado em *valor*, um parâmetro ortogonal criado para representar a importância de uma tarefa. Para que esse parâmetro efetivamente exiba um avanço, é necessário que o escalonador utilizado seja apropriado para essa tarefa. De um modo superficial, na teoria consta que cada tarefa possui uma função representando a utilidade gerada com o término dessa tarefa para o sistema, e a meta proposta para o algoritmo utilizado é maximizar a soma das utilidades acumuladas.

Apesar das qualidades exaltadas, essa técnica também possui suas deficiências. Estudos expuseram o proibitivo *overhead* trazido com a adição de funções ao algoritmo de escalonamento, requerendo muitas vezes poder computacional dedicado para essa tarefa. Além disso, é discutida a dificuldade encontrada na determinação de funções ou *valores* adequados, uma vez que o processo de escolha constitui uma base essencial da teoria.

Transpondo críticas e deficiências, o parâmetro *valor* tem sido usado com freqüência desde a sua criação nas mais diversas áreas de sistemas tempo real. Tomado como uma das formas mais simples de adicionar um comportamento flexível a um sistema, a sua eficiência e superioridade podem ser atestadas através de vários trabalhos nesta área

[6, 16, 21, 22].

A outra teoria abordada foi a política de escalonamento TAFT. Criada especificamente para permitir a construção de sistemas confiáveis que não dependessem dos piores casos para tempos de execução das tarefas, a premissa básica é a utilização de *TaskPairs*, compostos por uma *MainPart* e uma *ExceptionPart*.

Essa divisão ocorre de forma transparente para o escalonador, e delimita as responsabilidades de cada uma das partes envolvidas. A MP deve ser sempre executada, e a EP só entra em cena quando isso não for possível.

Restringindo a perda de *deadlines* dos TPs, a política TAFT permite que escalonamentos sejam criados com tarefas cujos WCETs são desconhecidos, e que escalonamentos eficientes sejam criados com tarefas cujos WCETs apresentam oscilações grandes e precisam ser substituídos por ECETs, sem perdas para o sistema. Comuns no TAFT, situações de sobrecarga transiente são prontamente resolvidas por um eficaz tratador de *ExceptionParts*, que garante a funcionalidade do sistema.

A utilização de um parâmetro representando importância de uma tarefa é discutida na política TAFT por Nett et al. [27] e Gergeleit [28], porém a estratégia mostrou-se incapaz de apresentar resultados promissores nesta área, e sua ineficiência é atestada por Becker [29] e Becker et al. [7]. Como correção para essa falha é sugerido a junção das duas teorias apresentadas, ou seja, o uso da política TAFT aliada ao o escalonamento baseado em *valor*. Essa proposta é levada a diante e avaliada ao longo deste trabalho.

Capítulo 3

Definição do Problema

Este capítulo apresenta o escopo do presente trabalho definindo os principais problemas abordados, a constar: (1) representação da importância de uma tarefa e (2) escalonamento dinâmico baseado neste parâmetro. O cenário sob o qual este problema é tratado conta com sistemas tempo real com elevada utilização de CPU e geralmente expostos a situações de sobrecarga.

De uma maneira ampla os objetivos almejados estão ligados a duas das mais evidentes falhas do escalonamento baseado em *deadlines*: a incapacidade de distinção entre urgência e importância das tarefas e a subutilização do processador, causada pelo pessimismo trazido com o uso do WCET nos testes de aceitação. Além de transpor essas deficiências, ainda é esperado que o sistema possibilite algum tipo de "controle" sobre a execução das tarefas, uma vez que o cenário proposto não permite a execução de todas elas.

3.1 Descrição Informal

A necessidade do discernimento da importância entre tarefas fica evidente em situações de sobrecarga, onde as decisões de alocação de recursos são cruciais, já que em outros casos todas as restrições temporais podem ser cumpridas. Nessas situações um escalonador tradicional, como o EDF, pretere a tarefa mais importante (a que trará mais benefícios para o sistema) em prol da mais prioritária (neste caso, a mais urgente, cujo deadline encontra-se mais próximo), diminuindo a serventia ou até inutilizando o sistema em certos casos.

A teoria do escalonamento baseado em *valor* visa suprir essa deficiência. Após vários estudos na área, conforme apresentado no capítulo anterior, concluiu-se que o uso do *valor* representa uma das maneiras mais simples de aumentar a flexibilidade de uma aplicação tempo real. Embora esta teoria apresente problemas enfrentados na prática, como o *overhead* introduzido pelo algoritmo de escalonamento (que muitas

vezes precisa ser executado em um processador auxiliar para tornar-se viável), um balanço geral demonstra que o uso do *valor* expõe uma série de benefícios, e sua adoção é uma tendência em sistemas tempo real atuais.

Outro problema bem conhecido na área de tempo real que afeta sistemas tradicionais, independentemente do uso do parâmetro *valor*, é a subutilização dos recursos. Causada em geral por métodos de garantia baseados em pior caso, a subutilização é uma desvantagem derivada implicitamente do formalismo necessário para garantir o funcionamento de um sistema. A política TAFT de escalonamento visa aumentar a utilização efetiva em sistemas fugindo de sobrecargas artificiais, através do emprego de tempos esperados de execução. O abandono da modelagem baseada no pior caso é compensada pela implementação de um mecanismo eficiente de tolerância a falhas. Simplificadamente, essas características são ostentadas através da troca de funcionalidade pela garantia da execução.

Por sua vez, um sistema seguindo o método de escalonamento TAFT tende a apresentar várias situações de sobrecargas (fato característico da política utilizada) e, embora provido de um mecanismo de tolerância a falhas, conserva a mesma incapacidade de discriminação entre urgência e importância de tarefas, demonstrada pelos demais métodos de escalonamento.

A proposta inicial de adição de importância a um *TaskPair*, sob a política TAFT, é feita por Nett et al. em [27] e Gergeleit em [28], onde sugere-se o emprego do parâmetro α. Esse parâmetro (descrito na seção 2.3.2) indica a probabilidade de um ECET representar com sucesso o tempo de execução real de uma tarefa, e através dele seria medida a prioridade do TP. O parâmetro α foi de fato empregado para essa finalidade em trabalhos posteriores, onde é também referenciado como α-quantil, e os resultados mostraram-no inadequado para a designação, conforme Becker em [29] e Becker et al. em [7]. O problema diagnosticado remete ao fato de que o parâmetro α-quantil é utilizado durante o teste de aceitação para o escalonamento, mas não é considerado na equação de designação de prioridades dos TPs.

Portanto a importância da tarefa é visível (ou considerada) somente quando ela adentra o sistema, e não quando o escalonamento é feito em tempo de execução. Isso propicia uma execução imprevisível, possuam as tarefas um valor α alto ou não, e deixa o sistema instável durante sobrecargas (justamente o contrário do que a definição de importância de uma tarefa deve fazer).

A substituição do parâmetro α para representar a importância das tarefas é discutida nos mesmos trabalhos que aferem sua ineficiência. Como solução é indicada a utilização do parâmetro *valor* em conjunto com um escalonador de segundo nível apropriado. Essa proposta é aqui estudada, implementada e posteriormente testada, a fim de ponderar sobre sua validade.

Em suma, este trabalho pode ser visto como um modo de unir a teoria do escalo-

namento através de *valor* com a política de escalonamento TAFT, resultando em um sistema dotado dos benefícios apresentados por ambas as partes.

3.2 Critérios de Avaliação dos Algoritmos

Para comparar a eficiência dos diferentes algoritmos de escalonamento foi utilizado o critério de utilidade acumulada (UA). Segundo sua definição formal, quando tarefas possuem o parâmetro *valor* e o processo de escalonamento é realizado através de TUFs, o critério de otimalidade é definido pela utilidade total acumulada do sistema ao fim da execução.

Como apresentado ao longo da seção 2.2, o funcionamento do critério UA é simples: ao término de cada execução de uma tarefa τ_i a utilidade total do sistema é incrementada pela utilidade dessa tarefa, que pode ser constante ou gerada pela TUF (tomando como parâmetro de entrada o tempo ao término da tarefa).

Para facilitar a comparação de diferentes conjuntos de dados, o critério de UA para cada algoritmo analisado é avaliado segundo a taxa de utilidade acumulada, ou *Utility Accrued Ratio*, calculada conforme equação 3.1:

$$\mathrm{UA}_R = \frac{\text{utilidade acumulada}}{\text{utilidade total do conjunto de tarefas}} \tag{3.1}$$

Um valor de UA_R de 1.0 significa que a utilidade total acumulada foi igual a máxima utilidade possível e, portanto, que todas as tarefas do sistema foram executadas dentro de suas restrições temporais. Esse resultado ocorre somente quando o sistema encontra-se com uma utilização menor ou igual a utilização máxima de 100%; quando situações de sobrecarga entram em cena a UA_R começa a cair e os melhores algoritmos são aqueles que obtêm valores mais próximos de 1.0.

Ainda que a utilidade acumulada seja o critério ótimo para a análise de algoritmos quando o parâmetro *valor* encontra-se presente na simulação, esta não é a única métrica adotada durante a análise dos resultados. Isso se deve ao fato da UA, ao final do período simulado, resumir-se apenas a um número, incapaz de transparecer o que realmente aconteceu durante a simulação.

Para tanto, o desempenho individual de cada tarefa executada é monitorado durante as simulações realizadas, principalmente no que diz respeito às perdas (ou cumprimentos) de *deadlines*. Através desse monitoramento é possível gerar gráficos individuais (por tarefa), propiciando uma análise mais profunda dos dados e facilitando a compreensão de cada evento.

Essa análise individual do comportamento de cada tarefa torna-se trabalhosa conforme o tamanho do conjunto de tarefas, e é uma prática não usada em trabalhos similares, que preferem métricas universais para o desempenho do sistema. Contudo

trata-se de um passo essencial para o completo entendimento dos resultados gerados, e de maneiras para modificá-los da forma desejada.

Ambas as métricas utilizadas são representadas através de gráficos ao longo do capítulo 5, referenciando os resultados encontrados.

3.3 Algoritmos de Escalonamento Avaliados

Esta seção apresenta os algoritmos de escalonamento utilizados durante a análise efetuada, juntamente com uma breve descrição de cada um. A escolha foi baseada em características próprias de cada escalonador e influenciada por trabalhos anteriores, para permitir a replicação de resultados. São eles:

EDF - *Earliest Deadline First* - Algoritmo clássico e tradicional do escalonamento dinâmico, apresenta um desempenho ótimo durante situações de *underload* (utilização inferior a 100%) e, por esse motivo, é utilizado como referência em praticamente todos os trabalhos nesta área. Definido em [8], o EDF atribui prioridades de acordo com o *deadline* absoluto das tarefas, sendo a tarefa de menor *deadline* absoluto a receber a prioridade mais alta. Além disso, seu uso em situações de sobrecarga serve para avaliação do "efeito dominó"[1], que chega a restringir ou até mesmo incapacitar sua utilização;

HVF - *High Value First* - Este algoritmo leva em conta somente o parâmetro *valor* para relacionar as prioridades entre as tarefas. Naturalmente, a tarefa com o maior *valor* recebe a maior prioridade. Embora se trate de um algoritmo dinâmico, por seguir as TUFs, pode apresentar um comportamento estático quando os *valores* são imutáveis (o que ocorre no uso de *step functions*, por exemplo);

HDF - *High Density First* - Algoritmo utilizado por Buttazzo et al. em [17], representa uma mistura dos dois algoritmos vistos anteriormente. A seleção de prioridades ocorre através da densidade V_i/c_i, onde V_i significa o *valor* e c_i o tempo restante no pior caso para a execução da tarefa T_i ser completada. A maior prioridade é dada à tarefa com maior densidade.

Com exceção do EDF, que não leva em conta o parâmetro *valor*, os outros dois algoritmos utilizam essa variável em seu processo de atribuição de prioridades. Como um dos critérios adotados para avaliação dos algoritmos é o UA, que considera a utilidade total do sistema, é esperado que os dois últimos algoritmos se sobressaiam durante

[1]Ocorre quando uma tarefa não cumpre o respectivo *deadline* e continua executando, "roubando" poder de processamento destinado a outras tarefas, que tendem dessa forma a também perder os seus deadlines. O processo então se repete, aumentando a carga do sistema gradativamente.

sobrecargas, onde nem todos os processos podem ser atendidos e decisões sobre qual tarefa deve ser executada são tomadas.

No entanto, adianta-se que os algoritmos citados não atenderam às demandas especificadas, pois ignoram *deadlines* perdidos pelas tarefas, tornando-se inadequados aos propósitos desejados. Desta forma, um novo método de asserção de prioridades foi então concebido e é introduzido na próxima seção.

3.3.1 Algoritmo *Dynamic Misses Based*

O algoritmo *Dynamic Misses Based* (DMB) é definido com o intuito de controlar o comportamento geral das tarefas em situações de sobrecarga, apresentando uma perda de desempenho gradativa e previsível. Durante o processo de escolha de prioridades, o DMB leva em conta o *valor* da tarefa e a taxa de *deadlines* perdidos até o instante atual. A equação 3.2 demonstra como são calculadas as prioridades das tarefas:

$$P_i = V_i \times (k_1 + k_2.MD_i) \tag{3.2}$$

onde P_i representa a prioridade, V_i o *valor* e MD_i a taxa de *deadlines* perdidos[2] da tarefa τ_i. Os parâmetros k_1 e k_2 são as constantes da equação. Nesta descrição um valor P_i alto representa uma prioridade alta, e vice-versa.

O termo MD_i da equação pode ter duas interpretações diferentes, por ser usado ora com tarefas normais, ora com *TaskPairs*. A interpretação tradicional é exatamente a sua definição literal, na qual ele representa a taxa de *deadlines* perdidos. Essa taxa pode ser calculada segundo a equação 3.3:

$$MD_i = \frac{releases(\tau_i) - (executions(\tau_i) - misses(\tau_i))}{releases(\tau_i)} \tag{3.3}$$

onde:

- $releases(\tau_i)$: número de liberações da tarefa τ_i;
- $executions(\tau_i)$: número de vezes que a tarefa τ_i foi executada;
- $misses(\tau_i)$: número de execuções da tarefa τ_i completadas após o *deadline*.

Resumidamente, subtrai-se o número de execuções com sucesso do número de vezes que a tarefa foi liberada e divide-se o resultado por esse número de liberações, obtendo-se o percentual de erros apresentados. Em um caso sem perdas, o número de execuções corretas será igual ao número total de execuções, resultando em um $MD_i = 0$. Essa fórmula se torna necessária pois tarefas normais podem apresentar liberações que foram ou não executas, e execuções que cumpriram ou não o *deadline*.

Quando aplicada a *TaskPairs*, no entanto, a definição lançada não faz mais sentido,

[2] *Missed Deadlines*.

já que não existem mais *deadlines* perdidos. Neste caso a transposição do objetivo do parâmetro é feita, e a equação 3.4 passa então a referenciar a taxa de *ExceptionParts* executadas:

$$MD_i = \frac{executions(\mathrm{EP}_i)}{releases(\mathrm{TP}_i)} \tag{3.4}$$

onde:

- $executions(\mathrm{EP}_i)$: número de execuções da *ExceptionPart*;
- $releases(\mathrm{TP}_i)$: o número de liberações do $TaskPair_i$.

Com a obrigação de um TaskPair de executar a MP ou a EP antes do respectivo *deadline*, o cálculo para a definição do desempenho de um TP torna-se muito mais trivial do que o visto anteriormente para uma tarefa normal. Nesse caso basta dividir o número de EPs executadas pelo número de TPs liberados.

Em situações onde não há violação aos requisitos temporais definidos, o algoritmo DMB se comporta exatamente como o HVF. Porém esses são geralmente os casos de carga baixa do sistema, cenário onde o EDF é ótimo e o uso de qualquer outro tipo de escalonador não faz sentido (considerando-se a busca pelos cumprimentos das restrições temporais).

Para exemplificar o comportamento do algoritmo é preciso verificar quais as variáveis que a equação de cálculo de prioridades 3.2 apresenta, assim como quais as conseqüências de suas variações. São elas:

V_i: indica o *valor* da tarefa τ_i. Um aumento ou diminuição nesse parâmetro causa igual efeito na prioridade da tarefa. Modificações podem ser feitas de maneira contínua ou discreta (dependendo, por exemplo, do formato da TUF);

MD_i: indica a taxa de perdas, calculada conforme as equações 3.3 e 3.4. Um aumento do valor desse parâmetro, representando perdas de *deadlines* através da impossibilidade de execução antes do limite de tempo estabelecido, cria um aumento de prioridade. Esse termo da equação só apresenta uma variação negativa caso um aumento prévio tenha sido noticiado, uma vez que o valor inicial apresentado é zero, e ela é incapaz de exibir valores negativos.

Para ilustrar o funcionamento do algoritmo, vejamos o caso de sistemas que empregam *valores* estáticos ou *step functions*. Com a primeira variável da equação imutável as diferenças começam a aparecer quando a carga do sistema ultrapassa seu limite, tarefas começam a perder *deadlines* e, conseqüentemente, a segunda variável da equação sofre modificações. Prioridades das primeiras tarefas a perder *deadlines* aumentam, e tarefas que possuíam a maior prioridade em um determinado momento podem não estar mais nesta posição após alguns eventos (e começar a perder *deadlines* também).

Essa inversão que ocorre com tarefas trocando de lugar na lista de prioridades persiste indefinidamente até a sobrecarga acabar, ou até que o limite de 100% de perdas seja atingido por algumas tarefas. Esse limite de 100% representa o máximo que o parâmetro MD_i pode assumir, significando que nesse ponto as tarefas estão sendo totalmente descartadas e que a fronteira superior de prioridade dada pela equação 3.2 com a primeira variável fixa foi alcançada. A partir desse momento só uma diminuição de carga pode causar futuras modificações.

Nos casos onde a variável V_i não é fixa, ou seja, TUFs sem restrições são usadas para representar o *valor*, a mudança de prioridades e as inversões das tarefas continuam a ocorrer, porém dessa vez dependendo dos dois parâmetros trabalhando em conjunto.

As constantes apresentadas na equação 3.2 também permitem aumentar o grau de controle no comportamento do algoritmo. A primeira delas, k_1, geralmente assume um valor igual a 1 e serve para que, enquanto nenhuma tarefa apresentar perdas, o resultado da equação de prioridades seja igual ao próprio *valor* V_i da tarefa. Seu uso com um valor igual a 0 fará com que o algoritmo só execute normalmente assim que todas tarefas apresentarem uma perda MD_i maior que zero. Já a segunda constante, k_2, tem o papel de definir o quanto a mudança na taxa de perdas das tarefas influenciará no aumento da prioridade das mesmas.

Com as constantes k_1 e k_2 assumindo valores iniciais iguais a 1, a equação 3.2 se transforma na equação 3.5, cujo comportamento é bem simples: enquanto tarefas não apresentarem perdas, o escalonamento ocorre normalmente segundo seu próprio valor V_i; variações na taxa de perdas MD_i possibilitarão que uma determinada tarefa atinja, no máximo, duas vezes a prioridade conseguida sem nenhuma perda.

$$P_i = V_i \times (1 + MD_i) \tag{3.5}$$

Esse modelo apresentado na equação 3.5 foi o utilizado para as simulações efetuadas na fase de testes.

Alguns detalhes adicionais a respeito do algoritmo DMB devem ser observados. O primeiro é que o comportamento descrito só ocorre quando *valores* apropriados são escolhidos entre as tarefas; essa questão é de suma importância, pois *valores* inadequados podem anular o efeito do termo $(k_1 + k_2.MD_i)$ da equação e resultar em um procedimento idêntico ao de outros algoritmos tradicionais, i.e. não representando avanço algum.

O outro detalhe é relacionado às perdas de *deadline*, que podem ser controladas através do parâmetro *valor*, mas não de forma absoluta nem individual, apenas em relação às outras tarefas do sistema. Portanto não é possível definir o número de vezes que uma tarefa específica vai perder seu *deadline*, somente a porcentagem de perdas que essa tarefa apresentará em relação às demais tarefas.

3.4 Variação da Carga do Sistema

É evidente que sob qualquer condição de *underload* comparações de desempenho entre algoritmos de escalonamento não fazem muito sentido, independente do quão abaixo do limite de 100% for essa utilização. Porém a partir desse limite a situação muda, e o contrário não pode ser dito, uma vez que diferentes valores de sobrecarga são capazes de modificar resultados, havendo diferenças entre uma "pequena" sobrecarga e uma "grande" sobrecarga. A fim de distinguir essas designações é preciso definir como calcular e medir a carga, ou sobrecarga, do sistema.

A carga em um sistema computacional com um conjunto definido de tarefas é geralmente calculada *a priori* pela *utilização nominal*[3]. A equação 3.6 mostra como efetuar o cálculo da utilização nominal NL_{system} de um sistema composto por n tarefas:

$$\mathrm{NL}_{system} = \sum_{i=1}^{n} \mathrm{NL}_i \tag{3.6}$$

onde NL_i representa a utilização nominal de cada tarefa τ_i. Esta utilização nominal individual é calculada através da equação 3.7, mostrada a seguir:

$$NL_i = \frac{WCET_i}{P_i} \tag{3.7}$$

sendo P_i o período da tarefa.

A utilização nominal, seja ela de uma tarefa ou de todo o sistema, pode ser calculada a qualquer instante, por se basear no WCET de cada tarefa. Essa utilização é impreterivelmente teórica, pois é improvável que um sistema real deva executar todas as tarefas em seu pior caso. Entra em cena então o conceito de *utilização efetiva*[4], que é calculada utilizando-se os valores reais de tempos de execução das tarefas de um sistema. Por esse motivo, a utilização efetiva só pode ser calculada após as execuções terem ocorrido, e, ao contrário da utilização nominal, que é constante para um mesmo sistema, pode variar ao longo da execução, conforme mudança dos tempos de execução monitorados.

A utilização efetiva é calculada usando-se as mesmas fórmulas da utilização nominal, apenas modificando o parâmetro WCET_i da equação 3.7 pelo tempo real médio de execução da tarefa τ_i.

Os conceitos de utilização nominal e utilização efetiva estão fortemente ligados à política de escalonamento TAFT, já que um de seus propósitos é não utilizar valores de WCETs para as tarefas. Como mostrado nos resultados apresentados no capítulo 5, nem sempre uma utilização nominal superior a 100% representa uma utilização

[3] *Nominal utilization* ou *nominal load*.
[4] *Effective utilization* ou *effective load*.

efetiva superior a 100%. Essa é uma das causa da subutilização de recursos quando o escalonamento é realizado através de métodos que utilizam o WCET: a política de admissão de tarefas recusa novas tarefas assim que uma sobrecarga nominal é detectada (ela pode ou não representar uma sobrecarga efetiva).

3.5 Metodologia Utilizada

Após estudo, descrição e formalização do problema enfrentado é exibida aqui a forma como será conduzida sua resolução.

O primeiro passo constitui na criação de um ambiente de simulação que permite a reprodução dos testes anteriores e realização de novas experiências. Durante a fase de recriação dos trabalhos o ambiente deve ser o mais fiel possível ao utilizado previamente. A descrição em detalhes dos parâmetros utilizados nesse ambiente é realizada ao longo do próximo capítulo.

Após a concepção do simulador propriamente dito é necessário realizar a transposição da teoria de escalonamento baseado em *valor* e da política de escalonamento TAFT, de modo que elas possam ser representadas no ambiente utilizado. Esse passo é realizado conforme trabalhos anteriores respectivos às duas áreas. Os algoritmos que serão utilizados, descritos na seção 3.3, além do novo algoritmo proposto, o DMB, são então implementados no ambiente de simulação.

Para a análise em situações de sobrecarga é variada da carga do sistema, nominal e efetiva, conforme discutido na seção anterior.

Na fase de testes os métodos utilizados são postos a prova através de uma avaliação objetiva, segundo o critério de utilidade acumulada, para que sua eficiência seja atestada e comprovada. Além disso o comportamento individual de cada tarefa é inspecionado para verificar se o domínio desejado da taxa de perdas através da atribuição do *valor* se concretiza. Todas essas métricas foram expostas na seção 3.2.

Capítulo 4

Projeto de Experimentos para Simulação

A simulação é uma ferramenta bastante utilizada na avaliação de escalonadores em geral, pois permite que resultados sejam obtidos com menor esforço e em muito menos tempo do que em uma implementação real. Embora os próprios resultados não possam ser considerados firmes o suficiente para sustentarem sozinhos a teoria que forma sua base, são usados como ponto inicial para uma futura implementação, poupando esforços desnecessários em diversos testes preliminares dispendiosos de serem construídos.

Com a finalidade de observar se o prognóstico desenhado ao longo deste trabalho realmente se concretiza, foram criados modelos de simulação dos algoritmos tradicionais (EDF, HVF, HDF) e do novo algoritmo proposto (DMB). A simulação dos mesmos foi realizada com o auxílio do software OMNeT++ [33]. O conjunto de tarefas a ser escalonado, bem como suas características, foram extraídos do *benchmark Hartstone*. Tais questões, assim como os demais detalhes considerados durante as simulações realizadas, são descritos nos subcapítulos seguintes.

4.1 *Benchmark* para Sistemas Tempo Real

O relatório técnico *Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications* (Hartstone: Requisitos para Benchmark Sintético de Aplicações Tempo Real Críticas) é apresentado em [32] por Weiderman e tem como objetivo definir uma série de exigências para testes de eficiência no escalonamento de tarefas tempo real críticas.

É defendido o fato de que a implementação de tais *benchmarks* é útil para avaliar o comportamento de algoritmos de escalonamento, protocolos, linguagens, compiladores e sistemas operacionais e apontar qual componente representa um empecilho para o contínuo aumento do desempenho do sistema. Esse processo é realizado com a repetição

dos mesmos *benchmarks* em ambientes totalmente idênticos, exceto pelo componente cuja avaliação é desejada.

O relatório descreve cinco definições de testes que podem ser utilizados em sistemas tempo real. É salientado que o uso do termo “tarefa” designa uma atividade, sendo um modo conveniente de referir-se a uma *thread* ou a um processo. Segue uma visão geral das categorias de testes propostos, em ordem de dificuldade:

Série PH: Testa o sistema com tarefas periódicas com freqüências harmônicas. É entendido por “freqüências harmônicas” que as tarefas devem ter um período regular, e a freqüência de cada uma deve ser um múltiplo inteiro de todas as outras tarefas com freqüências menores. A importância deste teste reside no fato de que freqüências harmônicas são encontradas facilmente em aplicações tempo real e são as mais fáceis de serem tratadas pelos algoritmos escalonadores;

Série PN: Essa série representa aplicações similares às da série PH, porém com as freqüências das tarefas compatíveis com requisitos da aplicação (freqüências naturais de fenômenos físicos, por exemplo) ao invés de requerimentos da implementação (freqüências exigidas pelo hardware ou detalhes da própria implementação ou linguagem);

Série AH: Relata um teste com uma série PH em segundo plano e outro conjunto de tarefas dirigidas a interrupção (aperiódicas) em primeiro plano. Representa sistemas que respondem a eventos externos. O objetivo é minimizar o tempo de resposta das tarefas aperiódicas ao mesmo tempo em os *deadlines* das tarefas periódicas não deixam de ser cumpridos;

Série SH: Representa um teste de uma série PH que necessita sincronização, introduzindo as possibilidades de bloqueio e inversão de prioridades. É útil também para investigar a eficiência de vários mecanismos de sincronização;

Série SA: Conjunto de tarefas que combina as características das últimas duas descrições. É o teste mais complicado e que demanda mais esforço do sistema, complexo a ponto de poder servir como protótipo de vários sistemas tempo real.

A Série PH, primeira apresentada e mais simples de todas, é descrita em detalhes durante o relatório. É composta por tarefas puramente periódicas e harmônicas, e pode representar, por exemplo, um programa monitor de vários sensores funcionando a taxas de atualização diferentes que apresenta os resultados sem a intervenção do usuário (ou necessidade de interrupção).

Inicialmente essa série é composta por cinco tarefas, cada uma com freqüências múltiplas das tarefas anteriores, e é descrita na tabela 4.1.

	Freqüência	Tempo de Execução
Tarefa 1	1 Hz	16 ms
Tarefa 2	2 Hz	8 ms
Tarefa 3	4 Hz	4 ms
Tarefa 4	8 Hz	2 ms
Tarefa 5	16 Hz	1 ms

Tabela 4.1: Conjunto inicial de tarefas da Série PH

Como requisito do *benchmark* temos que todas tarefas devem estar em fase, ou seja, devem iniciar ao mesmo tempo - o que representa o pior caso possível. Cabe ao escalonador decidir qual deve ser executada primeiro, sendo que o *deadline* de cada tarefa é igual ao seu período. É recomendado pelo relatório baterias de três execuções, cada uma com um tempo de duração que represente algo em torno de 30 segundos de execução.

O conjunto inicial de tarefas possui uma utilização nominal de 80%, e, portanto, não apresenta grandes dificuldades para escalonadores por prioridades dinâmicos. A fim de obter uma evolução nos testes, quatro experimentos distintos são apresentados com o propósito de aumentar gradativamente essa utilização nominal e possibilitar a observação dos resultados com o sistema sob cargas crescentes. São eles:

Experimento 1: A freqüência da tarefa 5 é aumentada em 8Hz a cada execução. Esse teste permite observar a capacidade do sistema de lidar com granularidades muito finas e de alternar rapidamente entre processos.

Experimento 2: A partir dos valores iniciais, as freqüências de todas tarefas são aumentadas em 10%, depois em 20% e assim por diante. Testa a capacidade do sistema de lidar com um aumento de carga gradativo porém balanceado.

Experimento 3: A partir dos valores iniciais, o tempo de execução de todas tarefas é aumentado em 1ms, depois em 2ms e assim por diante. Aumenta a carga do sistema sem modificar a freqüência das tarefas.

Experimento 4: Partindo-se do cenário inicial, novas tarefas com freqüência de 4Hz e tempo de execução de 2ms são adicionadas. Esse experimento visa testar a capacidade do sistema de lidar com um grande número de tarefas.

O restante do relatório técnico discute quatro possíveis implementações da Série PH na linguagem ADA, o que foge ao escopo deste trabalho.

4.2 Ferramenta de Simulação

O software escolhido para auxiliar a simulação realizada foi o OMNeT++[1], por ser fundamentalmente um ambiente de simulação de eventos discretos. Baseado em componentes, modular e com código aberto, o OMNeT++ fornece a base necessária para a criação e simulação de diversos tipos de sistemas. Criado especificamente para a simulação de redes de comunicação, é também usado com sucesso em outras áreas, como arquitetura de hardware e processamento paralelo, pelo fato de possuir uma arquitetura genérica e flexível.

O OMNeT++ provê arquitetura de componentes para a criação de modelos; os componentes (módulos) são definidos por uma linguagem de alto-nível própria, chamada linguagem NED, e programados em C++. A execução do modelo criado ocorre em cima de um núcleo de simulação, que pode ser facilmente acoplado gerando-se uma aplicação - binários que executam por padrão em uma interface gráfica, para melhor visualização dos resultados. A figura 4.1 mostra o software OMNeT++ durante uma das simulações realizadas.

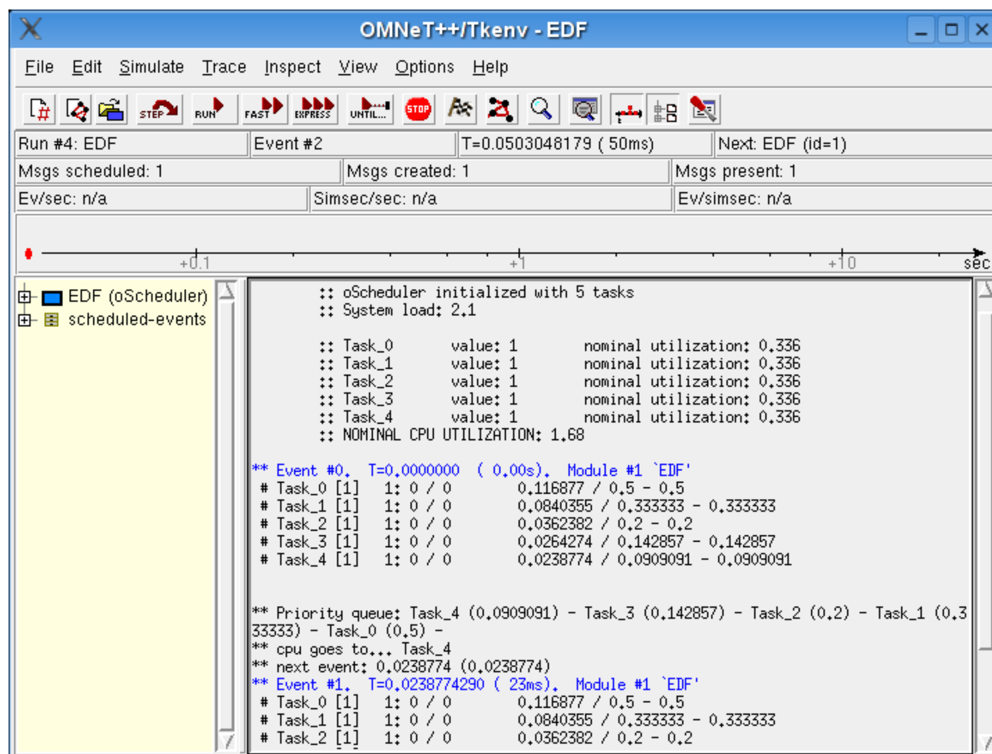


Figura 4.1: Interface gráfica do software OMNeT++

A definição de um simulador no OMNeT++ é baseada em componentes. Um componente pode, opcionalmente, possuir parâmetros, capazes de representar qualquer elemento da simulação como, por exemplo, o tempo de retardo presente na rede simulada. Os parâmetros podem ter seus valores determinados em sua própria definição

[1]OMNeT++ *Discrete Event Simulation System* versão 3.2.

ou através do arquivo de configuração da simulação, o `omnetpp.ini`. Esse arquivo de configuração é de grande valia, pois permite que diferentes dados de entrada sejam utilizados pelo mesmo arquivo executável, sem que seja necessária uma nova compilação (o que ocorreria, por exemplo, se a definição dos parâmetros fosse realizada durante a descrição dos módulos).

O arquivo de configuração de um simulador é dividido em opções gerais, como tempo total da simulação e nome do arquivo que irá registrar as estatísticas, e parâmetros da simulação, definidos de acordo com os objetivos traçados para aquela simulação específica. Há o conceito de *simulation runs*, que permitem a definição de diferentes valores para os mesmos parâmetros de um módulo, cada qual a ser utilizado em sua determinada *run*.

Além de tudo isso, o OMNeT++ é uma ferramenta acadêmica usada em projetos de diversas universidades e foi assunto de vários artigos científicos internacionais.

4.2.1 Modelagem

O primeiro passo para descrever um novo ambiente de simulação através do software OMNeT++ é a definição de uma topologia de rede, o que faz sentido, por se tratar de um software com esse objetivo. Essa descrição é feita na linguagem NED, com basicamente a relação dos nodos e das ligações entre esses nodos, e é usada para a representação gráfica (de trocas de mensagens, por exemplo) durante a simulação. Como este trabalho não tem por finalidade a representação de uma rede, não há uma definição de topologia, apenas a descrição de um módulo (o que pode ser visto, para fins de compatibilidade ou comparação, como uma rede de apenas um nodo). Esse módulo é representado pela classe escalonadora, a única que efetuará a comunicação com o núcleo OMNeT++.

Além da inclusão no arquivo de topologia, o escalonador precisa ainda implementar em C++ uma classe disponível pela API do OMNeT++ para tornar-se um módulo. A maneira mais simples de cumprir essa exigência é estender a classe designada para módulos simples, chamada `cSimpleModule`. Através de uma macro é feito o anúncio da classe como um módulo OMNeT++ e o acoplamento com o nodo de mesmo nome (definido no arquivo de topologia). O esqueleto de uma classe módulo é automaticamente gerado pelo OMNeT++, restando apenas a implementação possível de três métodos importantes da classe `cSimpleModule`:

`void initialize()`: método chamado apenas uma vez, durante a inicialização do simulador, pode ser visto e usado como construtor da classe;

`void finish()`: método chamado apenas ao término da simulação, é usado como um

tipo de destrutor, adequado para, entre outras tarefas, salvar as estatísticas coletadas ao longo da execução;

`void handleMessage(cMessage *msg):` executado sempre que uma mensagem for recebida, é o método principal da classe a ser implementada e contém a lógica interna do módulo com todas funcionalidades necessárias.

O método `handleMessage` trabalha em conjunto com métodos de envio de mensagem. Como o simulador construído não tem por objetivo envio e recebimento de mensagens para outros nodos (ou módulos), elas são utilizadas como maneira de controlar a passagem do tempo. Isso é realizado através do envio de mensagens de um módulo para si mesmo, executado pelo método `scheduleAt (simtime_t t, cMessage *msg)`, também implementado em `cSimpleModule`, onde o parâmetro `simtime_t t` define quanto tempo a mensagem levará para ser recebida. O uso agregado desses dois métodos permite um ambiente de execução orientado a eventos (mensagens) onde os tempos são rigidamente definidos.

É importante salientar que a API definida pelo OMNeT++ é muito ampla e flexível, e que foi dada apenas uma breve descrição dos meios utilizados na criação do simulador, não sendo objetivo em nenhum momento a exposição completa do software. Pelo fato de possuir código aberto o OMNeT++ transforma-se em uma ferramenta de simulação com inúmeras possibilidades e alternativas de uso.

A classe escalonadora é a única representada como módulo no OMNeT++. Todas as demais classes são escritas em C++ puro, o que dá flexibilidade ao simulador. A figura 4.2 mostra o diagrama de classes simplificado, representando apenas duas classes pertencentes ao OMNeT++ e as principais classes criadas no contexto desta dissertação.

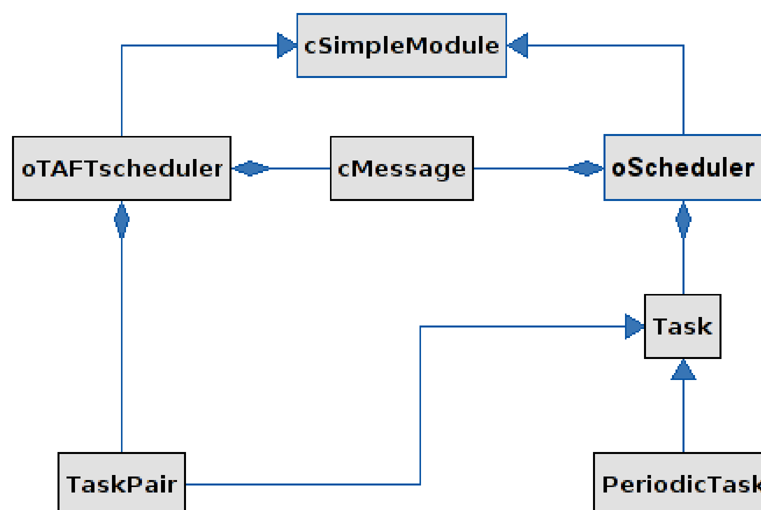


Figura 4.2: Diagrama de classes simplificado do simulador construído

Nessa figura é possível distinguir dois tipos de escalonadores: um normal e outro usado para seguir a política TAFT de escalonamento. Classes cujos nomes são precedidos pela letra '`c`' pertencem ao OMNeT++; nomes precedidos pela letra '`o`' significam módulos; classes sem precedência na nomenclatura representam arquivos normais C++. Segue uma breve descrição da função das principais classes envolvidas na simulação e presentes no diagrama de classes:

`cSimpleModule`: Classe base para todos módulos simples. Embora recheada de métodos, não efetua nenhuma operação, sendo preciso que uma subclasse a implemente para torná-la funcional;

`cMessage`: Classe de mensagens cujos objetos podem representar eventos, mensagens propriamente ditas, pacotes ou outra entidade na simulação. A capacidade que os objetos têm de serem agendados e entregues no próprio módulo em uma data futura é utilizada para o controle da passagem do tempo;

`oScheduler`: Subclasse de `cSimpleModule`, representa o escalonador utilizado para a simulação dos algoritmos tradicionais de escalonamento;

`oTAFTscheduler`: Subclasse de `cSimpleModule`, representa o escalonador utilizado para a simulação dos algoritmos analisados neste trabalho, seguindo a política de escalonamento TAFT;

`Task`: Interface para as classes que representam as tarefas do sistema. Contém as funcionalidades básicas necessárias para a simulação e é provida de estruturas de dados que armazenam estatísticas acerca da execução de uma tarefa;

`PeriodicTask`: Implementação de `Task`, representa tarefas periódicas normais, utilizadas nas simulações em conjunto com a classe `oScheduler`;

`TaskPair`: Implementação de `Task`, representa um *TaskPair*, utilizado nas simulações em conjunto com a classe `oTAFTscheduler`.

Deve ser ressaltado que, embora não exista a implementação explícita das classes de tarefas *MainPart* e *ExceptionPart*, o escalonador `oTAFTscheduler` enxerga objetos da classe `TaskPair` exatamente dessa forma, escalonando-os em dois níveis como se fossem duas tarefas distintas.

De modo a ilustrar a simplicidade e flexibilidade do simulador criado, pode-se empregar o exemplo de tarefas aperiódicas e esporádicas, não utilizadas nas simulações realizadas. Para que esses dois tipos de tarefa estejam presentes em trabalhos futuros basta que uma classe normal C++ as implemente, seguindo a interface `Task`.

Essa flexibilidade se estende para a implementação de novos métodos de escalonamento. A utilização de uma nova estratégia necessita somente a redefinição do

algoritmo de escolha de prioridades, o qual é realizado pelo método `getPriority(int policy)` da interface `Task`. Com isso é possível empregá-lo tanto como escalonador tradicional quanto como escalonador do segundo nível (encarregado pelas *MainParts*) da política TAFT.

Por fim, é necessário lembrar que outros arquivos C++ são gerados automaticamente pelo OMNeT++ para a compilação do arquivo binário da aplicação. Dentre esses arquivos temos código correspondente à topologia de rede e à tipos específicos de mensagens, se estas houverem sido especificadas. A listagem completa dos arquivos, assim como a implementação de todas as classes, pode ser encontrada no código fonte do simulador.

4.2.2 Coleta de Estatísticas

Outra característica que representa uma grande vantagem da simulação em relação à construção de um ambiente de execução real diz respeito à coleta de estatísticas. Em um sistema real essa coleta deve ser feita de modo concorrente com a própria execução do sistema, depende de dados indisponíveis ou às vezes difíceis de serem obtidos, consome poder computacional e memória. Por outro lado, em uma simulação esse processo é totalmente controlado e não conta com as desvantagens citadas.

Durante a criação e modelagem do simulador essa característica foi bastante explorada. Os componentes principais, como processador, escalonador e tarefas são completamente monitorados e, ao fim da execução, têm seus dados armazenados para futura análise, de onde são retirados os resultados obtidos. Além disso, para cada simulação é criado um arquivo .log, contendo os passos que foram executados e destacando, caso ocorram, a presença de erros na simulação.

4.2.3 Números Aleatórios

Para a geração de números aleatórios foram utilizadas funções providas pelo próprio OMNeT++. Por serem usados em simulações, esses números não são aleatórios "reais", mas sim produzidos através de algoritmos determinísticos. Essa característica é essencial para a reprodução dos resultados obtidos, tanto em situações idênticas como em situações onde há uma análise isolada de determinado componente do sistema. Nesse caso valores aleatórios diferentes a cada execução podem camuflar ou até mesmo sobrepor resultados originados pelo componente avaliado.

O OMNeT++ utiliza o conceito de "Geradores de Números Aleatórios" (*Random Number Generator* - RNG) para destacar a natureza determinísticas dos algoritmos. Um RNG utiliza uma semente inicial, responsável por gerar um número aleatório e a próxima semente. O processo, então, se repete. Dessa forma, iniciando-se com a mesma semente a mesma seqüência de números é gerada.

Embora o software permita a inserção de um RNG definido pelo usuário, através da interface `cRNG`, foi optado pela escolha de um algoritmo já existente, o *Mersenne Twister*. Esse algoritmo foi criado por Matsumoto e Nishimura [34], apresenta um período de $2^{19937} - 1$ e é bastante rápido.

Durante os testes realizados, cada tarefa simulada foi ligada a um RNG e uma semente próprios. Isso significa que, independente do algoritmo utilizado, uma dada tarefa τ_i apresentará sempre a mesma seqüência de números aleatórios. Exemplificando: se em uma bateria de testes (i) a tarefa τ_i apresentar 150 execuções, e em outra bateria posterior (ii) apenas 120, os 120 primeiros valores de números aleatórios observados serão exatamente iguais em ambas as baterias (i) e (ii), independente de qualquer fator ligado à qualquer outra tarefa. Todo esse cuidado é imprescindível para evitar que valores aleatórios distintos sejam capazes de, por exemplo, deturpar resultados e atribuir ao escalonador utilizado as diferenças encontradas.

4.2.4 Funções de Distribuição de Probabilidades

Levando-se em conta que a estratégia TAFT foi criada para tratar variações no tempo de execução das tarefas por trabalhar ECETs é incoerente realizar os testes através do tempo de pior caso presente no modelo inicial. Becker em [29] sugere que durante a simulação os tempos de execução variem entre 50% a 100% do WCET de cada tarefa (variação considerada conservadora). Além disso, através de observações é notado que o comportamento dos tempos de execução das tarefas pode ser aproximado por duas Funções de Distribuição de Probabilidade (*Probability Distribution Function* - PDF): $beta(2,3)$ e $uniforme$.

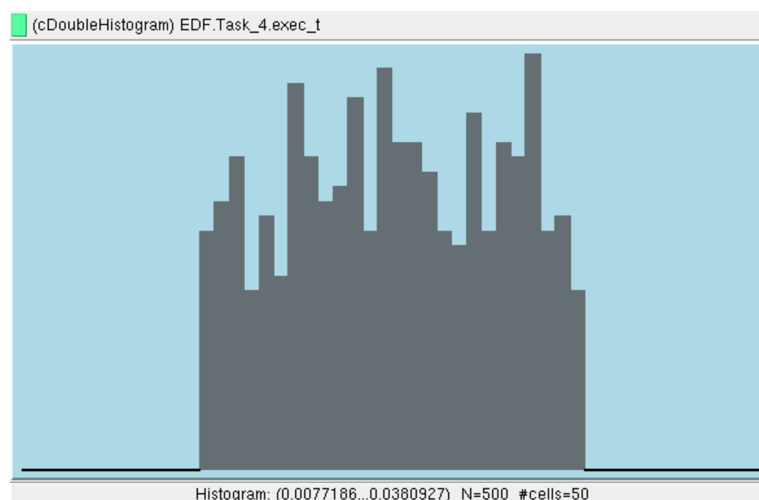


Figura 4.3: Distribuição de valores segundo a PDF $uniforme$

O uso da PDF $uniforme$ se dá por tratar-se de uma função amplamente utilizada em experimentos de simulação. É caracterizada por distribuir regularmente os valores

dentro da faixa determinada, com uma média de 0,5 em relação aos valores mínimo e máximo. A figura 4.3 mostra um histograma gerado automaticamente pelo software OMNeT++, e atualizado constantemente conforme a execução da simulação. Este histograma representa a distribuição dos tempos de execução de determinada tarefa durante um evento da simulação, após 500 amostras coletadas.

Já a PDF *beta*, utilizada com parâmetros $a = 2$ e $b = 3$, apresenta um comportamento aparentemente mais próximo do obtido na prática. Seus valores encontram-se distribuídos mais juntos da média, que no caso é de 0,47, e têm menor probabilidade de estarem próximos dos valores mínimo ou máximo definidos. A figura 4.4 ilustra o histograma dos tempos de execução de uma tarefa após 500 observações, e pode ser visto que utilizando-se a função $beta(2, 3)$ os tempos dificilmente estarão perto do pior caso.

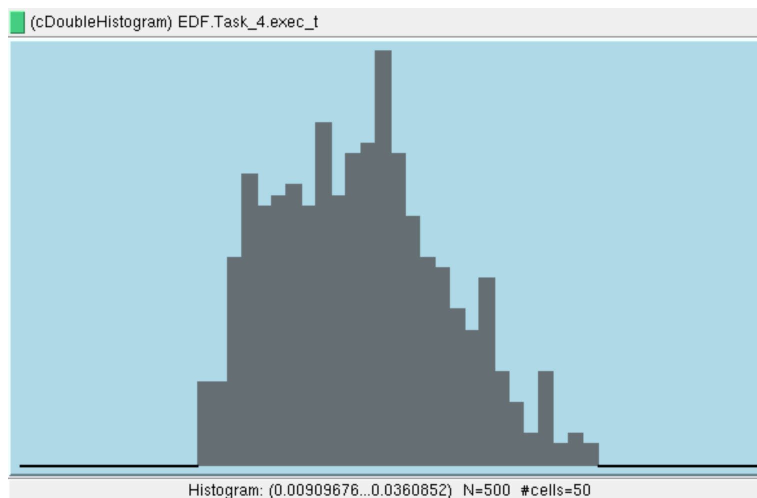


Figura 4.4: Distribuição de valores segundo a PDF $beta(2, 3)$

4.2.5 Conjunto de Tarefas

São dois os principais motivos que levaram ao uso das definições lançadas pelo *benchmark Hartstone* no escopo desta dissertação: o primeiro se baseia no relatório em si, adaptado para sistemas tempo real e com testes definidos e detalhados; o segundo motivo é relativo aos trabalhos anteriores sobre a política de escalonamento TAFT, pois simulações baseadas neste *benchmark* foram realizadas por Becker em [29] e por Becker et al. em [7]. Com isto é garantido um alto grau de transparência para a simulação, além da possibilidade de replicação e continuação dos resultados prévios. Para manter a coerência com os testes anteriores, empregaram-se as mesmas escolhas de séries de tarefas e métodos de aumento de utilização efetiva.

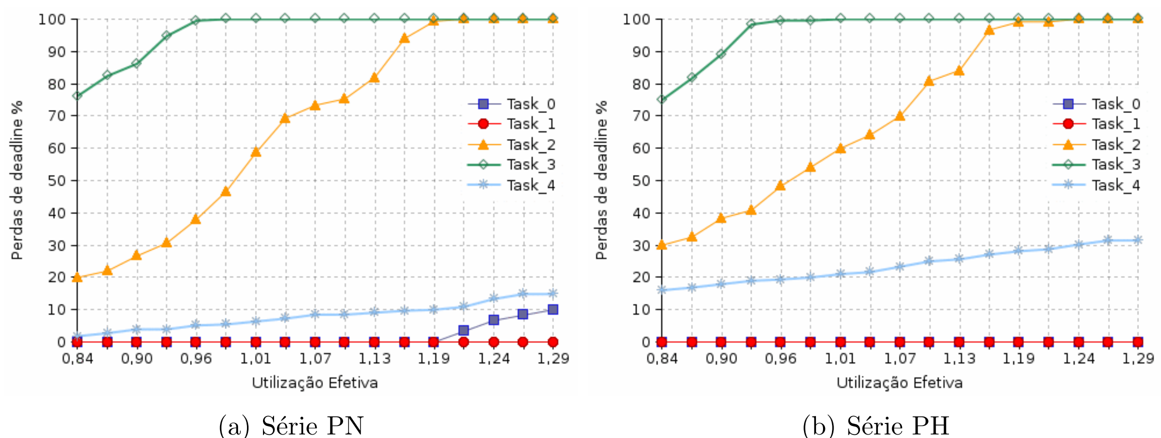
Como os objetivos deste trabalho não abrangem tarefas aperiódicas nem visam a utilização de sincronismo entre tarefas, foi mantida a utilização das séries PH e PN.

Os dois modelos estão descritos na tabela 4.2.

	PH Series		PN Series	
	Freqüência	WCET	Freqüência	WCET
Task_0	1 Hz	160.00 ms	2 Hz	80.00 ms
Task_1	2 Hz	80.00 ms	3 Hz	53.28 ms
Task_2	4 Hz	40.00 ms	5 Hz	32.00 ms
Task_3	8 Hz	20.00 ms	7 Hz	22.85 ms
Task_4	16 Hz	10.00 ms	11 Hz	14.54 ms

Tabela 4.2: Séries PH e PN: freqüência e tempo de execução no pior caso de cada tarefa

As séries possuem a mesma utilização nominal em sua configuração inicial, e a diferença básica consiste nas tarefas com períodos harmônicos, presentes na série PH e ausentes na PN. A mudança de conduta gerada por essa diferença pode ser vista na figura 4.5, onde o mesmo cenário de simulação é executado alterando-se apenas a série de tarefas utilizada.



(a) Série PN

(b) Série PH

Figura 4.5: Diferentes comportamentos das duas séries de tarefas utilizadas

A explicação para os resultados distintos das figuras 4.5(a) e 4.5(b) recai exatamente na mudança do parâmetro período de cada tarefa. De um modo geral, porém, pode-se ver um comportamento bastante similar por parte das duas séries.

Como os modelos de tarefas da tabela 4.2 são utilizados normalmente durante simulações com algoritmos tradicionais, adaptações foram necessárias para que eles fossem usados nas simulações que envolvem a política TAFT de escalonamento. Nesses casos as tarefas descritas transformaram-se em *TaskPairs*, com a *MainPart* possuindo o tempo de execução no pior caso correspondente ao modelo, e a *ExceptionPart* apresentando um WCET relativo a 5% da WCET da MP. Deve-se ressaltar que esse tempo de execução aparentemente curto atribuído às EPs vem justamente ao encontro das definições da política TAFT, que afirma que uma EP deve possuir funcionalidade mínima e servir apenas para deixar a aplicação em um estado seguro.

Embora todos os testes realizados tenham sido feitos com as duas séries especificadas, o modelo PN foi escolhido como a principal série, pelo fato de apresentar um comportamento mais próximo de tarefas reais, e é a série utilizada para representar os resultados deste trabalho.

4.2.6 Carga do Sistema

Em se tratando dos procedimentos para o aumento da utilização efetiva sugeridos, é definido que a análise da troca de contexto e admissão de novas tarefas não vêm ao encontro dos objetivos primários da simulação (embora possam representar testes significativos para o futuro). Assim sendo, os experimentos 2 e 3 apresentados na seção 4.1 são os únicos passíveis de utilização. Os experimentos sugerem o crescimento gradual da carga através do aumento das freqüências das tarefas (experimento 2) e aumento do tempo de execução das tarefas (experimento 3). Esse último procedimento foi o escolhido. No entanto os tempos de execução de todas tarefas foram escalonados de forma percentual, e não absoluta, conforme sugerido pelo benchmark. Assim, a cada aumento de carga a utilização nominal do sistema também é aumentada pelo mesmo percentual.

A base para a carga do sistema é definida pelo conjunto inicial de tarefas descrito pelo *benchmark Hartstone* e apresentado na tabela 4.2, totalizando uma utilização nominal de 80%. Essa utilização nominal inicial é expandida em até 230%, através do aumento do tempo de execução no pior caso de cada tarefa por esse percentual. Um aumento de 230% em cima dos dados iniciais gera uma utilização nominal teórica do sistema de 184%, limite superior utilizado.

No entanto esse limite nominal só é alcançado caso sejam considerados os piores casos dos tempos de execuções das tarefas, prática não adotada durante as simulações. Para exemplificar o limite efetivo alcançado podemos usar o caso da PDF $uniforme$, que gera valores com uma média de 0,5 entre os limites superiores e inferiores. Com a variação dos tempos das tarefas entre 50% e 100% do WCET, o uso da função $uniforme$ representa uma média de 75% para os WCETs gerados nas simulações. Desse modo o valor máximo de utilização nominal, 184%, se transforma em uma utilização efetiva teórica correspondente à 75% disso, ou seja, 138%.

Embora o cálculo inicial de definição dos parâmetros de entrada seja feito através da utilização nominal das tarefas, os resultados são analisados segundo a utilização efetiva encontrada. Em geral é registrada uma diferença de até 30% entre esses dois valores, e esse é um dos motivos da subutilização de sistemas tradicionais tempo real. Essa diferença pode ser confirmada no gráfico da figura 4.6, que ilustra de um modo geral as utilizações efetivas das simulações, discernindo-as segundo a PDF empregada.

É importante notar que a utilização efetiva média do sistema só ultrapassa a linha

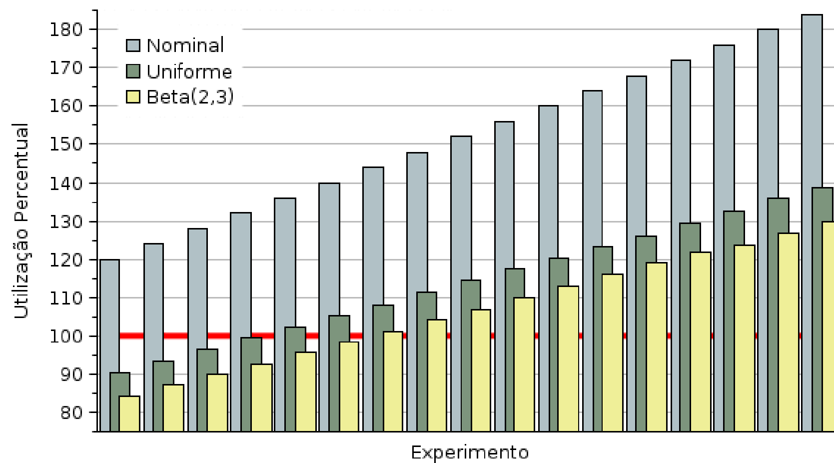


Figura 4.6: Diferenças entre utilizações nominal e efetiva

dos 100% quando a utilização nominal já está acima de 130% (no caso da utilização da função $beta(2,3)$, que apresenta uma média um pouco menor, o valor nominal já se encontra acima dos 140%). Pode-se notar aqui que a estimativa de limite efetivo para a PDF $uniforme$, que foi de 138%, é bastante próxima do valor encontrado na simulação.

4.3 Considerações Finais

Utilizou-se OMNeT++ como ferramenta de simulação principalmente por ser este um software voltado para a simulação de eventos discretos e por ser bastante flexível. Desse modo o controle da passagem do tempo ficou ao cargo do núcleo de simulação, enquanto todas as funcionalidades e os algoritmos necessários foram implementados como classes C++ comuns. A coleta de estatísticas também ficou por conta do OMNeT++, que possui estruturas de dados especiais para isso, como vetores e histogramas. Esses dados podem ser visualizados graficamente durante a execução e também gravados para um tratamento futuro, através das próprias ferramentas que o OMNeT++ provê ou de qualquer software adequado para esse encargo.

Além dos próprios algoritmos de escalonamento, mais três características importantes do OMNeT++ sofreram variações:

Modelo de Tarefas: conforme apresentado, os modelos de tarefas utilizados podem ser constituídos por tarefas harmônicas (série PH) ou não-harmônicas (série PN);

Função de Distribuição de Probabilidades: responsável pela definição dos tempos de execução reais de cada tarefas, as funções utilizadas foram $beta(2,3)$ e $uniforme$;

Carga do Sistema: a fim de simular condições de sobrecarga, a carga do sistema é constantemente alterada ao longo dos testes.

Os diferentes algoritmos descritos no capítulo anterior são avaliados individualmente, escalonando tarefas em um único nível, e como escalonadores de segundo nível da política TAFT. Foram feitos testes utilizando diversos modelos de escalonamento através da variação dos dados de entrada conforme o *benchmark Hartstone*. Cada uma das simulações executa diversos *simulation runs* com valores diferentes para o WCET de cada tarefa (simulando a carga total do sistema), a fim de analisar as perdas causadas pela crescente utilização. Todos experimentos duraram um período equivalente a 30 segundos simulados.

Os parâmetros gerais da execução são definidos no arquivo de configuração e são fixos: tempo máximo de execução, sementes dos RNGs, nomes dos arquivos com dados estatísticos, etc. Os parâmetros variáveis da configuração também foram definidos em arquivos de configuração, porém cada *run* possui o seu arquivo próprio. Essa característica do OMNeT++ proporcionou a criação de um único binário (simulador em si), e a realização dos testes consistiu em alimentar esse simulador com diferentes arquivos de configuração (dados de entrada).

Os dados estatísticos são coletados individualmente em cada execução (*run*), e são gravados em arquivos como texto. Após a simulação a análise é realizada pelas ferramentas de tratamento de vetores, chamada *plove*, e de tratamento de histogramas, chamada *scalars*. Ambas as ferramentas acompanham o software OMNeT++ e foram criadas para essas finalidades. As demais análises estatísticas que se fazem necessárias são realizadas através de planilhas e gráficos construídos especificamente para isso.

Capítulo 5

Resultados Obtidos

As simulações realizadas estão divididas em duas partes. A primeira apresenta a comparação dos algoritmos descritos na seção 3.3 com o acréscimo do algoritmo desenvolvido, o DMB. Essa comparação é realizada de forma semelhante aos testes de trabalhos anteriores, que incluíam o TAFT (Becker em [29]) ou somente os demais algoritmos de escalonamento descritos (Buttazzo et al. em [17]). Durante essa fase de simulações os *valores* das tarefas são mantidos constantes.

A segunda parte é baseada na avaliação do DMB em situações de sobrecarga transiente em conjunto com o TAFT. O objetivo desta simulação é avaliar se uma estratégia de mudança dos *valores* das tarefas, efetuada ao longo da execução, repercute nos resultados esperados para esse parâmetro e se permite também algum tipo de domínio sobre a taxa de execução de EPs para cada TP.

5.1 Avaliação dos Algoritmos com *Valores* Constantes

Nesta seção são apresentados os resultados obtidos da análise dos algoritmos selecionados, de uma forma bastante similar à encontrada em trabalhos prévios. Com um conjunto de tarefas e uma PDF escolhidos o sistema é simulado sob diferentes condições de carga, começando abaixo da marca de 100% e aumentando gradativamente, a fim de observar o desempenho de cada algoritmo nessas circunstâncias.

A distinção feita é que cada algoritmo é primeiramente analisado trabalhando de forma individual e mais tarde como escalonador do segundo nível da política TAFT (encarregado do escalonamento das *MainParts*). Os *valores* atribuídos a cada tarefa (ou *TaskPair*) não sofrem modificações ao longo dos experimentos, e podem ser vistos na tabela 5.1.

Os resultados são exibidos conforme discussão feita na seção 3.2: seguindo a avaliação tradicional e objetiva realizada pela comparação de desempenho entre os algo-

Tarefas		*Valor*
Task_0	TP_0	V_0
Task_1	TP_1	$1.500 \times V_0$
Task_2	TP_2	$0.850 \times V_0$
Task_3	TP_3	$0.750 \times V_0$
Task_4	TP_4	$1.062 \times V_0$

Tabela 5.1: Tarefas utilizadas nos experimentos e respectivos parâmetros *valor*

ritmos de escalonamento através de sua taxa de utilidade acumulada (UA_R), como demonstrado na equação 3.1. Após isso são apresentados os desempenhos individuais de cada algoritmo, com gráficos contendo o comportamento peculiar de cada tarefa segundo sua taxa de *deadlines* perdidos. Essa taxa foi definida na seção 3.3.1, com a equação 3.3 referindo-se a tarefas normais e a equação 3.4 a *TaskPairs*.

5.1.1 Algoritmos Tradicionais

Embora uma análise similar tenha sido feita por Buttazzo et al. em [17], exceto pelo algoritmo DMB, a realização de novos experimentos ocorre devido a três razões principais:

- simulação de um conjunto de tarefas mais próximo de uma aplicação real, com parâmetros específicos para sistemas tempo real ao invés de valores aleatórios;
- mudança de foco para a utilização efetiva ao invés da utilização nominal;
- análise do desempenho individual das tarefas ao longo da simulação, além do desempenho geral.

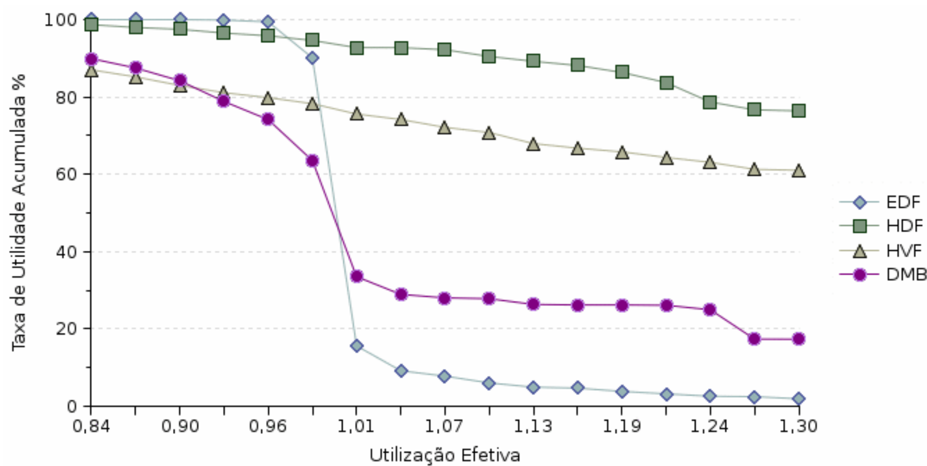


Figura 5.1: Desempenho geral dos escalonadores através da UA_R

O gráfico da figura 5.1 mostra a taxa de utilidade acumulada dos quatro escalonadores durante os experimentos. Como esperado, o algoritmo EDF consiste na solução

mais eficiente antes da barreira de 100% de utilização, mas tem um queda bruta no desempenho após esse momento. O conhecido "efeito dominó" causa um impacto muito grande e transforma o EDF em uma solução impraticável durante sobrecargas.

A utilização efetiva, conforme observado, aumenta ao longo do tempo em que a simulação está sendo executada (ao contrário da utilização nominal, que é constante), pois os tempos de execução apresentam mudanças para cada instância de tarefa escalonada. Essa é a razão pela qual é possível perceber o algoritmo EDF perdendo alguns deadlines, mesmo com uma utilização efetiva abaixo da linha dos 100%. Em algum ponto da simulação a utilização efetiva esteve acima da média calculada - e acima dos 100%.

A figura 5.2 mostra o desempenho individual das tarefas sob o escalonador EDF. Observando o gráfico é possível concluir que o procedimento de escalonamento é realizado na base do "tudo ou nada", ou seja: quase não há perdas de *deadline* no período anterior à utilização efetiva de 100%, e quase não há cumprimento de *deadlines* após esse ponto. É importante notar que as tarefas são prejudicadas uniformemente com o advento da sobrecarga e do conseqüente efeito dominó.

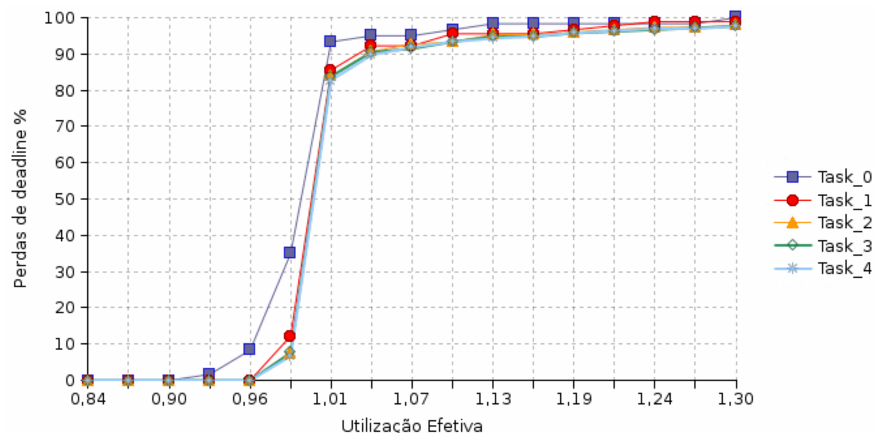


Figura 5.2: Comportamento individual das tarefas: EDF

De volta à figura 5.1, percebe-se que o DMB é outro algoritmo que sofre com o impacto da sobrecarga, embora um pouco mais brandamente do que o EDF. Seu desempenho após a barreira de utilização de 100% é melhor pois é a utilidade acumulada que está sendo avaliada, e o algoritmo DMB utiliza o *valor* em sua equação de designação de prioridades (parâmetro ignorado pelo EDF). A figura 5.3 mostra o desempenho individual das tarefas sob o DMB. O fato positivo é o reconhecimento da degradação uniforme apresentada por todas tarefas, como proposto na especificação do algoritmo. Outra característica evidente é a relação entre a taxa de degradação das tarefas e seu *valor*.

Os outros dois algoritmos analisados, HVF e HDF, não demonstram uma queda de desempenho considerável durante a execução da figura 5.1, aparentando imunidade à

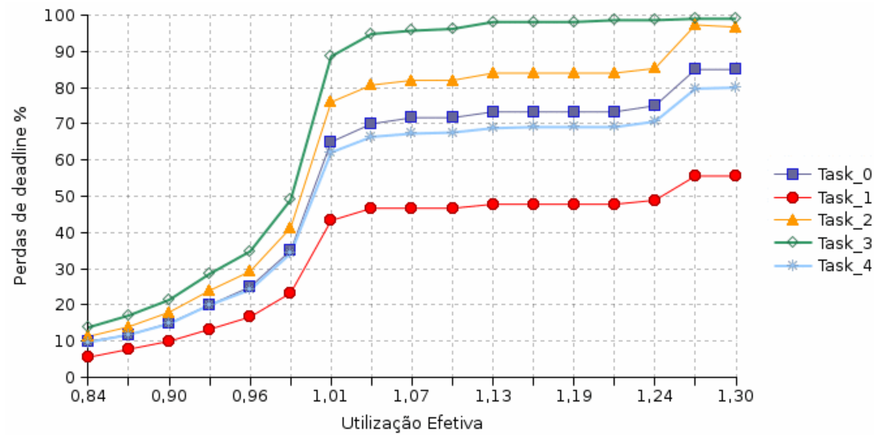


Figura 5.3: Comportamento individual das tarefas: DMB

sobrecarga e dando a impressão superficial de um melhor desempenho. A explicação para esse fato é simples: enquanto os algoritmos EDF e DMB tentam executar todas tarefas, mesmo quando isso não é possível, HVF e HDF começam a descartar tarefas com as menores prioridades (segundo suas métricas) conforme ocorre o aumento da utilização do sistema, criando dessa forma espaço para a execução das tarefas remanescentes.

Observando a figura 5.4, que mostra o comportamento das tarefas sob a tutela do escalonador HDF, fica claro que a tarefa `Task_0` é a única a perder *deadlines*, até o instante em que ela é totalmente descartada. Posteriormente a tarefa `Task_1` inicia um processo parecido, com perdas de *deadlines* que aumentam muito rapidamente até ser também completamente descartada. Ao longo da simulação as três tarefas remanescentes, que apresentam certamente as três maiores densidades, sofrem muito pouco com o aumento da carga do sistema.

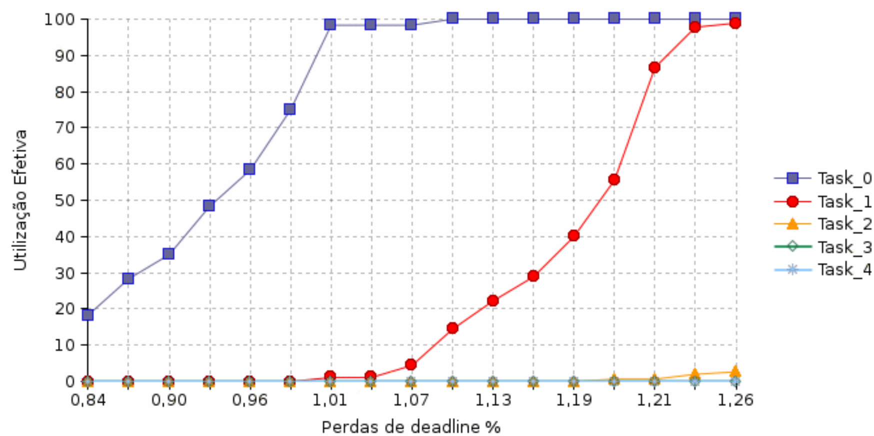


Figura 5.4: Comportamento individual das tarefas: HDF

O gráfico da figura 5.5 exibe o comportamento desenvolvido pelo algoritmo HVF. De uma forma geral ele pode ser comparado ao comportamento do escalonador HDF,

exceto pelo fato das tarefas sacrificadas diferirem de um gráfico para outro. Neste caso temos `Task_3` e `Task_2`, obviamente providas dos menores *valores* dentre as tarefas do conjunto, como as primeiras no processo de descarte de tarefas.

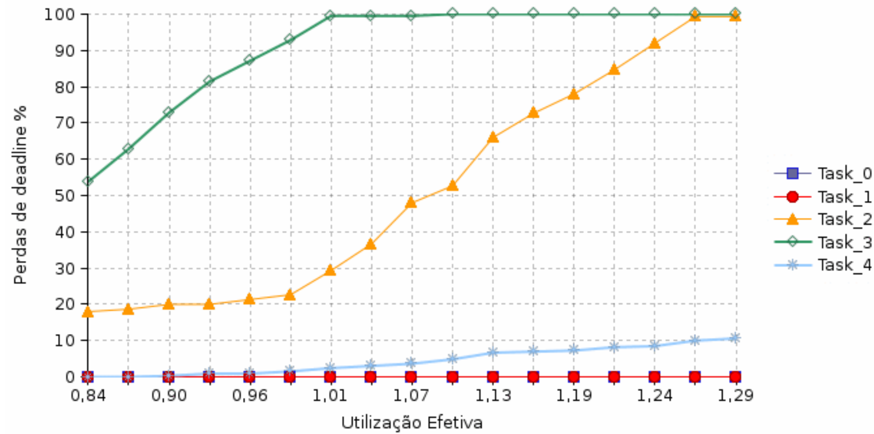


Figura 5.5: Comportamento individual das tarefas: HVF

O comportamento distinto apresentado pelos quatro algoritmos pode ser esclarecido com uma análise mais profunda de seus modos de funcionamento. Algoritmos como o EDF e o DMB atuam como os escalonadores de prioridade dinâmica que são. Enquanto isso o HVF, embora também dinâmico, age como um escalonador de prioridade fixa. O responsável por esse comportamento são os *valores* constantes utilizados durante a simulação, preservando as mesmas prioridades entre as tarefas do início ao fim, o que pode ser visto como uma situação onde todas tarefas possuem *value functions* iguais a uma constante. O algoritmo HDF, devido ao seu funcionamento, é um algoritmo dinâmico em nível de *job*[1], por considerar o tempo de execução restante em sua equação de escalonamento (vide seção 3.3), mas, com *valores* constantes, mudanças de prioridade ocorrem raramente. Com tudo isso as tarefas com as menores densidades (e conseqüentemente menores prioridades) são as que sofrem as maiores perdas durante toda a simulação.

5.1.2 Escalonador TAFT

Nesta subseção o mesmo tipo de análise previamente explicada é repetido, com a diferença de que aqui os algoritmos são usados em conjunto com o escalonador TAFT. Com a ausência de *deadlines* perdidos na política TAFT, os gráficos individuais de cada escalonador apresentam o percentual de *ExceptionParts* executadas, uma vez que no melhor caso todas *MainParts* são executadas e esse percentual é de 0%. Caso o oposto ocorra (nenhuma execução de *MainPart* a tempo) todas *ExceptionParts* serão executadas, gerando um valor de 100% no gráfico.

[1] *Job-level dynamic-priority.*

Devido à essa mudança de parâmetros durante o acompanhamento individual, não é possível fazer nenhuma relação direta e absoluta entre os valores dos gráficos representando os resultados das diferentes simulações mostradas nas seções 5.1.1 e 5.1.2. Todavia é possível analisar os comportamentos individuais das tarefas em cada escalonador, como segue.

A figura 5.6 mostra o desempenho dos quatro algoritmos em conjunto com o escalonador TAFT. É importante observar que, para uma melhor visualização do gráfico, o eixo y inicia com um valor de 50%, ao invés do valor de 0% usado anteriormente.

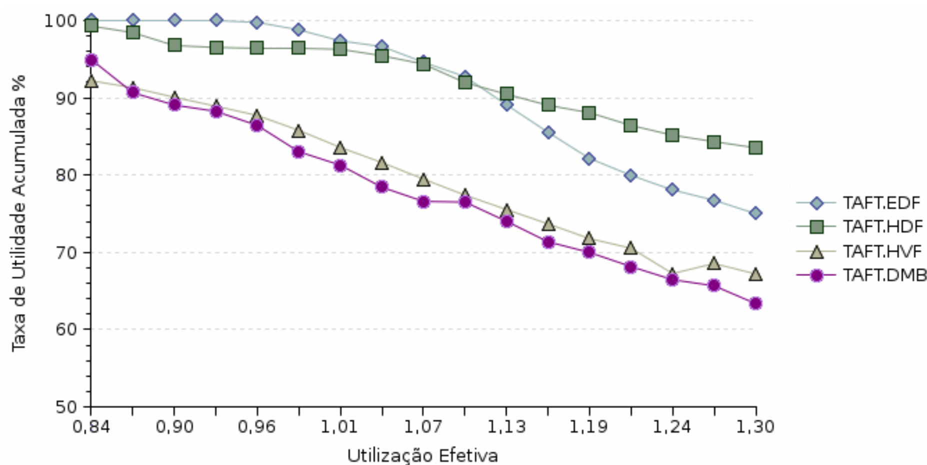


Figura 5.6: Desempenho geral sob a política TAFT através da UA_R

Sob a política de escalonamento TAFT um cenário muito diferente é obtido, com uma maior taxa de utilidade acumulada geral em todos algoritmos e sem quedas bruscas de desempenho. A razão para que esses resultados apareçam está na concepção de escalonamento do TAFT, transformando tarefas normais em *TaskPairs*. Como visto na seção 2.3, antes do deadline de um *TaskPair* uma de suas partes (*MainPart* ou *ExceptionPart*) deve ser executada. Logo, não existem *deadlines* "perdidos" como acontece em escalonadores tradicionais, mas sim execuções de *ExceptionParts*, o que é algo totalmente diferente e não pode ser comparado diretamente com a perda de um *deadline*. Sem desrespeitos às restrições temporais sobrecargas transientes não são carregadas adiante na execução, e futuras tarefas não são prejudicadas.

É possível novamente notar os excelentes resultados obtidos pelo EDF no início da simulação, que só são superados em torno da marca de 110% de utilização efetiva - desta vez, porém, sem a queda brusca de desempenho. Além disso podemos ver que o HDF apresenta o melhor resultado geral em condições de sobrecarga, enquanto o DMB e o HVF demonstram uma degradação de desempenho similar ao longo do tempo.

Os resultados individuais para cada tarefa quando se utiliza o TAFT com o escalonador proposto por Becker em [29], o EDF, podem ser vistos na figura 5.7. O comportamento exibido assemelha-se ao observado nos algoritmos HDF e HVF da seção

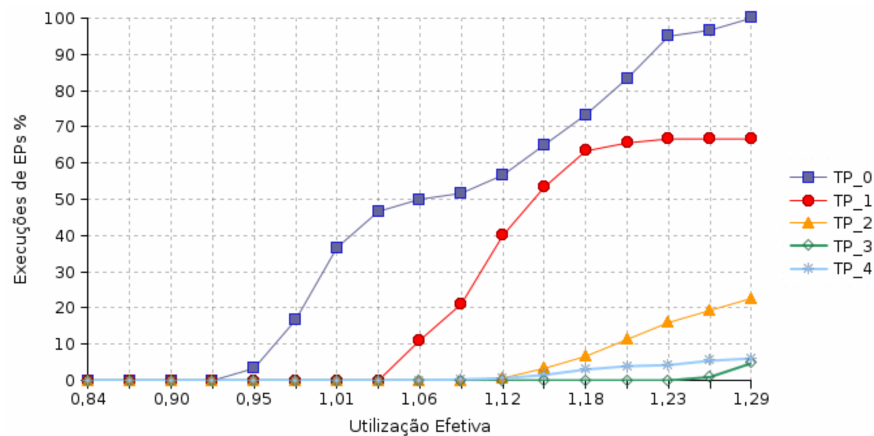


Figura 5.7: Comportamento individual das tarefas: EDF em conjunto com o TAFT

anterior, onde tarefas começam a ser descartadas assim que a sobrecarga é detectada. No entanto desta vez é possível observar múltiplos *TaskPairs* apresentando execuções de *MainParts* e *ExceptionParts* ao mesmo tempo (perdas balanceadas), como pode ser visto na parte final da simulação. Isso ocorre em contraste com o gráfico anterior, onde tarefas de prioridades menores apresentavam cada vez uma taxa de perdas maior, com aumento gradativo e isolado, até serem descartadas, para só então outra tarefa começar a apresentar uma taxa de perdas crescente (perdas desbalanceadas).

Fica claro que sem o efeito dominó - mérito da política TAFT - o EDF se tornou, senão o melhor algoritmo, uma escolha plausível para situações de sobrecarga.

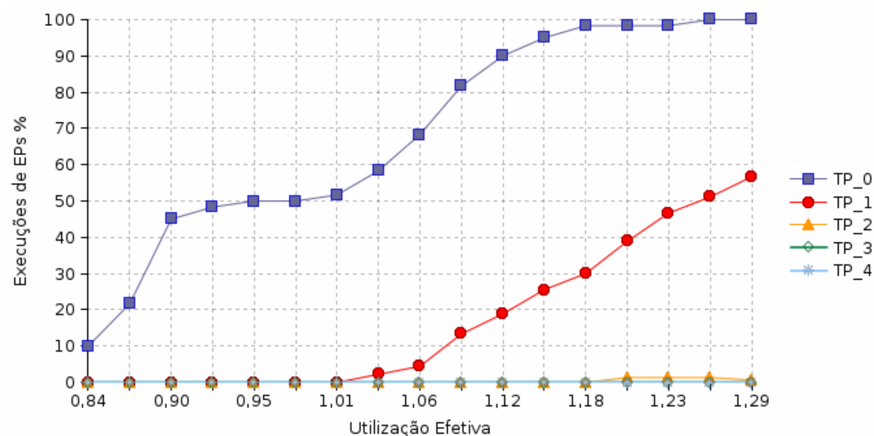


Figura 5.8: Comportamento individual das tarefas: HDF em conjunto com o TAFT

Os resultados individuais do escalonador HDF são apresentados na figura 5.8. Seu desempenho conserva-se semelhante ao observado na análise anterior, onde ambas as tarefas com as menores densidades executam *ExceptionParts* durante a simulação, em porcentagens diferentes, enquanto as outras três mostram praticamente nenhuma perda durante os experimentos. A única diferença verificada é semelhante ao observado no caso do EDF: vários intervalos apresentam múltiplos TaskPairs executando *MainParts*

e *ExceptionParts* simultaneamente.

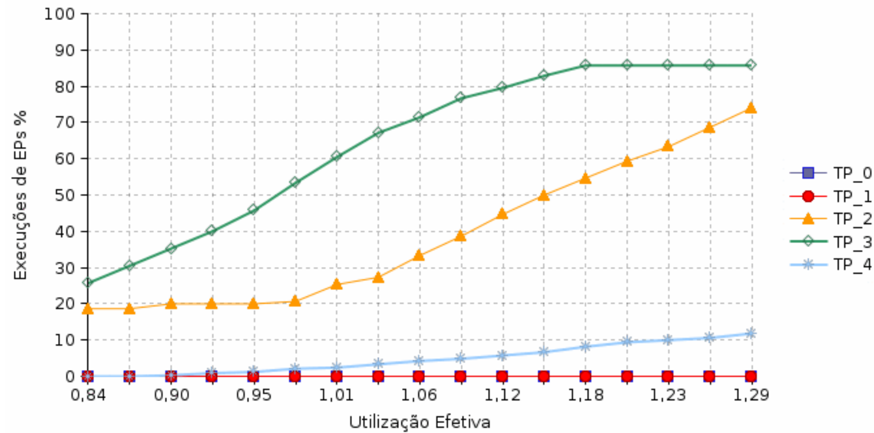


Figura 5.9: Comportamento individual das tarefas: HVF em conjunto com o TAFT

O gráfico correspondente ao escalonador HVF, presente na figura 5.9, mostra-se bastante semelhante ao gerado sem a política TAFT (vide figura 5.5). Com um comportamento geral preservado, as mesmas tarefas que antes possuíam altas taxas de perdas de *deadlines* agora apresentam altas taxas de execuções de EPs.

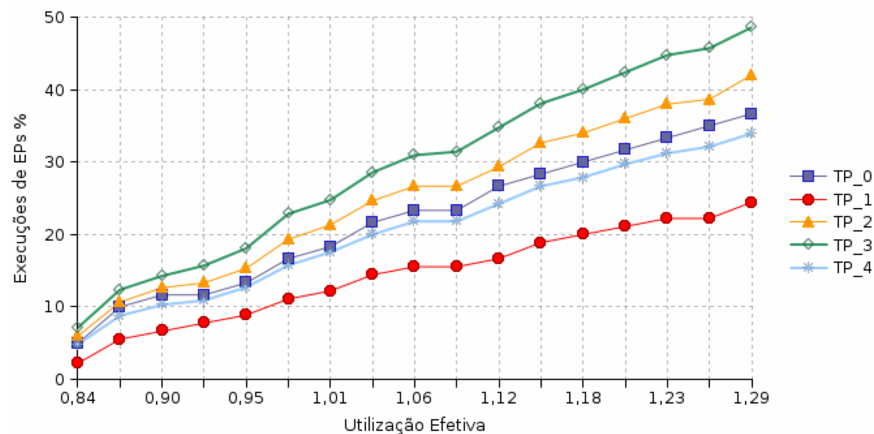


Figura 5.10: Comportamento individual das tarefas: DMB em conjunto com o TAFT

De fato, o algoritmo DMB aliado à política TAFT alcançou os resultados mais promissores em se tratando do objetivo traçado de balanceamento de perdas durante sobrecargas. Os resultados ilustrados pela figura 5.10 mostram os *TaskPairs* com um aumento linear da porcentagem de execução de *ExceptionParts*. Para uma melhor análise do gráfico a escala do eixo y é limitada pelo valor máximo de 50%, ao invés do valor prévio de 100% usado em gráficos similares.

Analisando-se dois *TaskPairs* distintos sob o escalonamento proporcionado pelo DMB em conjunto com o escalonador TAFT é possível estabelecer que a relação entre o número percentual de execuções de *ExceptionParts* é inversamente proporcional à relação entre os valores dos *TaskPairs*. Essa característica é encontrada entre quaisquer

dupla de *TaskPairs*, o que significa exatamente que o número total de execuções de *ExceptionParts* não pode ser controlado, mas o percentual de execuções de EPs de um *TaskPair* em relação à outro pode.

5.1.3 Resumo

Após todas análises gerais e individuais, é observado que, durante situações de sobrecarga, algoritmos tradicionais se comportam de uma das seguintes formas:

a) através de uma seqüência definida, uma tarefa de cada vez é escolhida para ser descartada em um taxa gradual, de modo que as tarefas restantes não sofram as conseqüências da sobrecarga - provavelmente mantém o sistema reativo[2], mas sem algumas funcionalidades;

b) tenta "ignorar" a sobrecarga e continua a executar todas tarefas normalmente - provavelmente perdendo vários *deadlines* e tornando o sistema menos reativo ou até mesmo não-funcional.

A adição do parâmetro *valor* tem um papel importante em ambos os casos descritos. Na situação **a)**, como mostram os resultados dos algoritmos HVF e HDF, a ordem em que as tarefas são descartadas (ou param de responder ao sistema) pode ser determinada pela escolha do *valor*. Na situação **b)**, como mostram os resultados do DMB, o uso adequado do *valor* determina a relação entre as perdas apresentadas pelas tarefas.

Com a adição da política TAFT existe também a vantagem relacionada à ausência da propagação de uma sobrecarga transiente: soluções como o algoritmo DMB podem satisfatoriamente manter o sistema em um estado um pouco menos reativo durante um período de tempo e imediatamente recuperar-se para o estado normal após isso. Esse comportamento satisfaz o objetivo de ter um sistema permanentemente sob alta utilização e que não apresente sérios riscos ou danos após uma sobrecarga temporária.

Outros algoritmos relacionados ao parâmetro *valor*, como o HDF, podem ser usados igualmente, com a desvantagem de perder as tarefas de menor prioridade, mas com o benefício de obtenção do melhor resultado geral. O algoritmo HDF é propício para situações onde tarefas possuam uma ordem hierárquica e algumas delas não devem executar EPs, salvas situações extremas onde não é possível evitar que isso ocorra.

[2] *Responsive.*

5.2 Avaliação do DMB-TAFT com *Valores* Dinâmicos

A continuação do trabalho realizado aponta para a consideração do uso de *valores* dinâmicos durante a execução, junto com a estratégia que apresentou, para as métricas avaliadas, os resultados mais satisfatórios: o algoritmo DMB aliado à política TAFT. O propósito fundamental dessa experiência é tentar aferir a capacidade do sistema de controlar prioridades de tarefas e serviços de maneira *on-line*.

Até o presente momento as simulações contaram somente com *valores* pré-definidos antes mesmo do início da execução, que não são modificados em nenhum instante. Uma vez que esses *valores* utilizados podem ser representados por *step functions*, a primeira alternativa para a inclusão de variações seria o uso de qualquer outro tipo de função, seguindo o modelo de TUFs. Funções diversificadas adicionam *valores* que oscilam em nível de *job*, representando a utilidade agregada ao término da execução em determinado instante de tempo.

Embora atinja o objetivo de adição de *valores* dinâmicos à simulação, o uso de TUFs irrestritas é descartado aqui por dois motivos principais. O primeiro remete à complexidade computacional introduzida por esse tipo de função durante o processo de escalonamento, deficiência citada por Davis et al. em [18] e McElhone em [3] - este último trabalho sugere como alternativa exatamente o uso de *step functions*, uma das razões pela qual a primeira simulação foi realizada dessa maneira.

O segundo motivo é que o comportamento em busca da maximização da utilidade, sob a óptica das TUFs, não é exatamente o comportamento esperado com a adição de valores dinâmicos. Por ocorrer em nível de *job*, a otimização das tarefas sob TUFs é realizada de forma similar em dois superperíodos distintos, tendendo a exibir uma constância com o decorrer do tempo. As prioridades oscilam igualmente em duas liberações distintas, e a organização das tarefas é feita visando um maior ganho no instante seguinte.

A reação desejada com a incorporação de *valores* dinâmicos assemelha-se um pouco ao conceito de "modos" ou "alternativas" de execução de uma tarefa, utilizados por Burns em [6], onde são descritas tarefas cujas funcionalidades podem ser realizadas com um grau determinado de qualidade, conforme a alternativa escolhida. Cada uma delas possui uma utilidade relacionada ao término de sua execução, e a maximização da utilidade ocorre quando as melhores alternativas de cada tarefa são escolhidas e finalizadas.

Este trabalho foca o conceito de "modo de execução" de uma maneira diferente, com o conceito referindo-se ao modo de execução do sistema ao invés do modo de execução de uma tarefa específica. Um exemplo que pode ser dado é de um sistema em

sobrecarga que necessita executar um serviço crítico, mas, em certo ponto da execução, este serviço é modificado. Tarefas responsáveis pelo serviço desatualizado devem perder *valor* para não serem mais executadas e tarefas responsáveis pelo novo serviço devem ganhar *valor* a fim de se tornarem essenciais para o sistema. Embora o sistema e as tarefas continuem os mesmos, a funcionalidade crítica, ou principal do sistema, foi alterada.

Os resultados obtidos são gerados com base no seguinte cenário: um conjunto determinado de tarefas está passando por uma sobrecarga transiente, com utilizações efetivas sempre maiores do que 100%. Esse conjunto é escalonado segundo a política TAFT, utilizando o algoritmo DMB como escalonador de segundo nível. A PDF utilizada é a $beta(2,3)$, o conjunto de tarefas é representado pela série PN e a simulação é observada durante 20 intervalos de tempo.

TaskPair	***Valor* Inicial**	***Valor* Final**
TP_0	V_0	V_0
TP_1	$1.50 \times V_0$	$1.50 \times V_0$
TP_2	$0.85 \times V_0$	$3.00 \times V_0$
TP_3	$0.75 \times V_0$	$3.50 \times V_0$
TP_4	$1.25 \times V_0$	$1.25 \times V_0$

Tabela 5.2: Variação ocorrida nos parâmetros *valor* ao longo dos experimentos

Os *valores* das tarefas, assim como a modificação que eles sofrem ao longo do tempo, podem ser observados através da tabela 5.2, que mostra os *valores* com que as tarefas começam a execução e os *valores* apresentados ao fim da execução. Inicialmente o parâmetro *valor* de todas as tarefas é fixo durante todo o experimento para ilustrar o que ocorre com o sistema caso nenhuma atitude seja tomada.

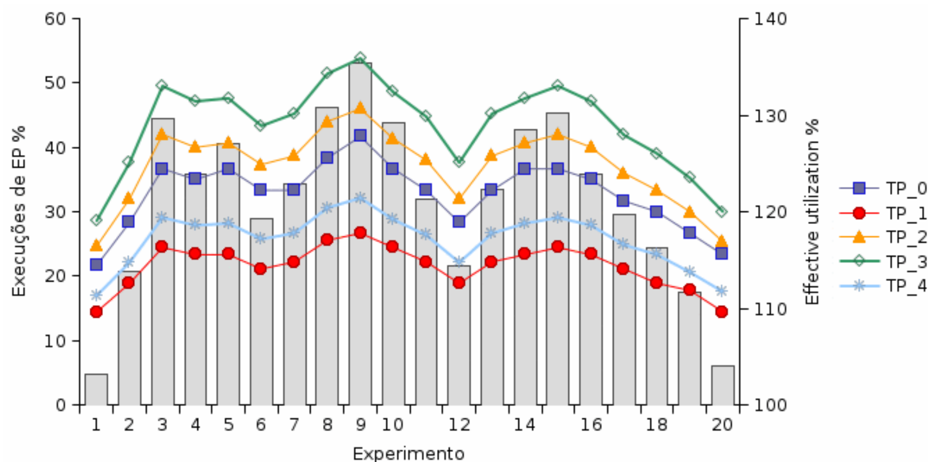


Figura 5.11: Utilização efetiva e percentual de execuções de *ExceptionParts*

Dois gráficos são fundidos na figura 5.11: o eixo y à direita mostra a oscilação da utilização efetiva durante o período simulado, representado pelas barras verticais; o

eixo y à esquerda mede o percentual de execução de *ExceptionParts*, representadas pelas linhas. A sobreposição dessas duas estatísticas permite a confirmação visual de uma afirmação feita na seção 3.3.1: o número absoluto de EPs executadas não pode ser determinado previamente, pois depende da utilização do sistema. Fica claro que o desempenho das tarefas oscila exatamente conforme a oscilação da utilização efetiva, porém a relação da taxa de execuções de EPs entre as tarefas é constante ao longo do gráfico.

É preciso lembrar que esse cenário inicial não apresenta mudança alguma nos *valores* das tarefas. O objetivo durante a execução é exatamente essa alteração, através de algum princípio definido, de forma a diminuir o percentual de execução de EPs de dois *TaskPairs* determinados: `TP_2` e `TP_3`. Logo, os dois *TaskPairs* que apresentam os piores desempenhos no início da simulação devem ser os que apresentam os melhores ao final dela.

5.2.1 Modos Definidos de Execução

A descrição de modos de execução consiste na forma mais simples de resolução do problema. É considerado que o sistema possui uma quantidade limitada de modos, cada qual provido de *valores* fixos para cada uma das tarefas. Para simplificar o funcionamento, esta simulação conta apenas com dois modos distintos. Os *valores* iniciais do sistema, exibidos na tabela 5.2, representam o primeiro modo, cujos resultados são mostrados até a metade do gráfico presente na figura 5.12 e são os mesmos vistos previamente na figura 5.11. Nesta primeira metade, as tarefas `TP_2` e `TP_3` possuem *valores* menores do que as demais tarefas, e com isso prioridades inferiores e um maior índice percentual de execução de EPs.

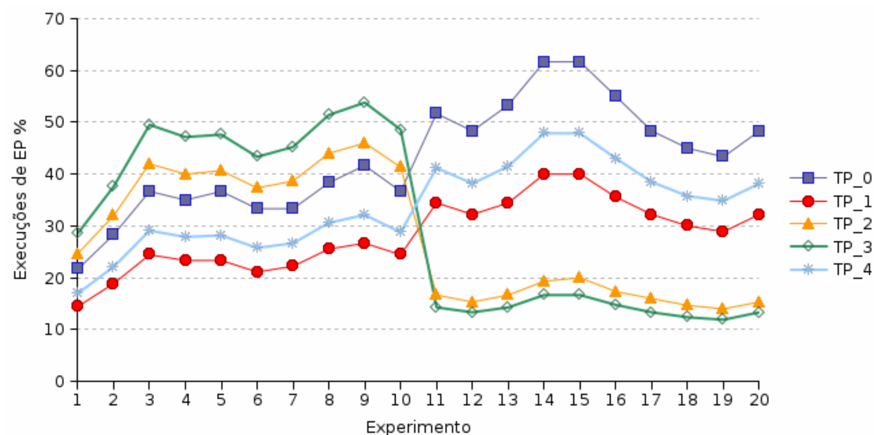


Figura 5.12: Simulação segundo dois modos definidos de execução

Exatamente na metade da simulação é então acionado o segundo modo de execução, correspondente aos *valores* finais da tabela 5.2, onde as tarefas mais prioritárias são

exatamente TP_2 e TP_3. Nesse modo os valores dessas duas tarefas são os maiores dentre o conjunto todo, com TP_3 sendo o *TaskPair* mais importante do sistema. Os três *TaskPairs* restantes mantêm seus *valores* originais. Essa mudança atômica de *valores* das tarefas causou um distúrbio abrupto da taxa de execução das EPs, claramente visível no gráfico.

É importante notar que, conforme já explicado, a variação absoluta no desempenho de cada tarefa é dependente da utilização efetiva do sistema naquele determinado momento. Como a utilização efetiva da simulação realizada oscila a todo instante (porém sempre acima dos 100%), há também uma oscilação no desempenho de cada tarefa. No entanto, todas tarefas oscilam de forma sincronizada umas em relação às outra, exceto ao longo do trecho específico onde ocorre a mudança de seus *valores*.

Com esse exemplo simples é possível mais uma vez atestar a capacidade do algoritmo DMB de controlar o comportamento individual de tarefas em condições de sobrecarga, realizando ajustes no parâmetro *valor*.

5.2.2 Mecanismo de Predição

Os resultados da seção anterior são considerados esperados e condizentes com o objetivo proposto. Porém é visível o choque que o sistema sofre com a mudança entre modos definidos de forma prévia, o que causa uma alteração brusca na funcionalidade de todo sistema. Além disso, modos pré-estabelecidos limitam o comportamento do sistema apenas às definições criadas. A fim de promover um comportamento mais suave e adaptativo para o sistema entra em cena o conceito do componente "monitor", utilizado para predizer a taxa de perdas de cada tarefa.

O monitor é encarregado de garantir que os modos de execução desejados sejam respeitados. Porém, em vez da especificação propriamente dita dos *valores*, ele pode lidar com restrições ligadas às tarefas. Por exemplo: determinado modo de execução pode conter a restrição de que tarefas τ_1 e τ_2 não devem ultrapassar 20% de perdas de *deadlines* no modo γ, ao invés de conter números representando os *valores* dessas tarefas para esse modo. Por meio dessas restrições e baseando-se em observações e dados passados, o monitor efetua o ajuste necessário dos *valores* correntes.

Além de não deixar o sistema preso à modos definidos de execução, o mecanismo de predição traz a vantagem de que o ajuste das prioridades, e conseqüentemente a mudança do comportamento do sistema, pode ocorrer de uma forma gradual. A figura 5.13 mostra esse mecanismo sendo utilizado na mesma situação anteriormente resolvida através da abordagem de modos de execução.

Observando-se os resultados presentes nos primeiros e nos últimos instantes de tempo do gráfico fica claro que os *valores* iniciais e finais das tarefas são os mesmos da simulação anterior. Isso significa que houve a mesma variação de desempenho (ou

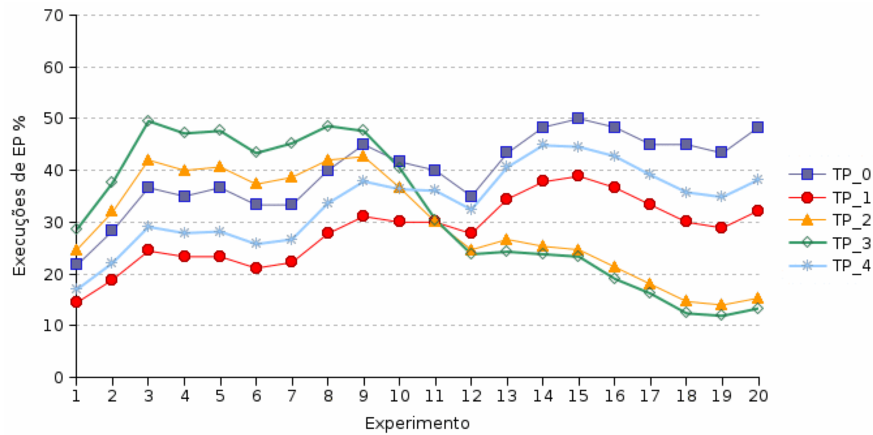


Figura 5.13: Simulação através do uso de um componente monitor

variação de comportamento) nos dois casos. No entanto, aqui o mecanismo de predição suavizou a mudança abrupta observada anteriormente. Observando-se mais a fundo, é possível notar a ação do monitor ter efeito no conjunto de tarefas do intervalo de tempo 8 até o intervalo de tempo 17.

5.2.3 Definição de uma Tarefa Crítica

Após o estudo de duas técnicas diferentes de alteração dinâmica do parâmetro *valor* das tarefas durante a simulação, nesta seção é demonstrado que o algoritmo DMB pode ser usado para obter-se um comportamento semelhante ao de outros algoritmos. A situação apresentada é a mesma das duas simulações anteriores, porém dessa vez o *TaskPair* `TP_2` é definido como uma atividade crítica que, enquanto possível, não deve perder deadlines (nesse caso, apresentar execuções de EPs).

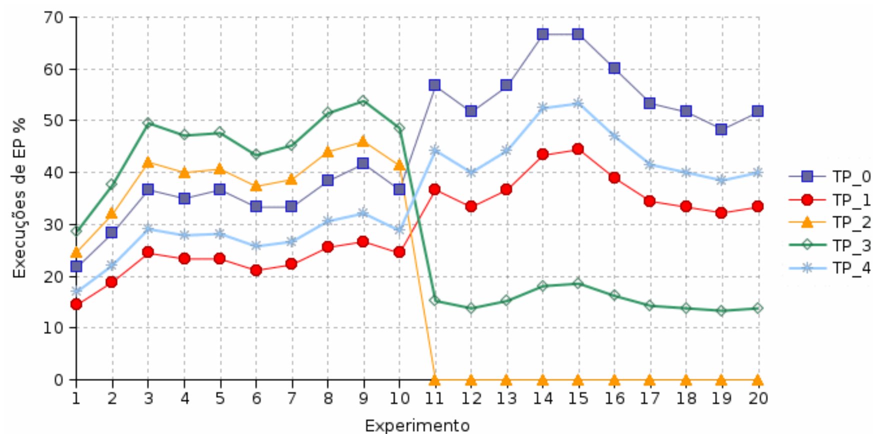


Figura 5.14: Simulação considerando-se uma tarefa crítica

O gráfico da figura 5.14 mostra o resultado da simulação efetuada, bastante semelhante aos resultados obtidos com o uso de modos definidos de execução (vide figura

5.12). Todavia, o *TaskPair* TP_2, agora considerado crítico, assume um *valor* de grandeza maior do que os *valores* das demais tarefas - nessa simulação o *valor* nominal assumido é 100. Conseqüentemente esse *TaskPair* não recai na execução de EPs, enquanto o restante do conjunto mantém o comportamento ordenado observado por gráficos anteriores.

Essa definição de tarefas "críticas" pode ser ampliada para grupos de tarefas: determinados grupos são criados através da atribuição de *valores* de diferentes grandezas para diferentes grupos. Dessa forma tarefas do grupo mais prioritário não apresentarão perdas, a menos que as tarefas dos outros grupos já estejam sendo totalmente descartadas. Mesmo assim, ao começar a apresentar perdas, as tarefas de um mesmo grupo continuarão seguindo o comportamento ordenado definido pelo parâmetro *valor*.

5.2.4 Síntese do Uso de *Valores* Dinâmicos

Embora os resultados obtidos previamente com o uso de *valores* constantes possam ser considerados promissores, é através da união com o escalonador TAFT e da mudança dinâmica de *valores* que o algoritmo DMB mostra realmente seu potencial, sob a métrica pré-estabelecida de controle da degradação do sistema.

Utilizando-se dos dois métodos de modificações de *valores* propostos, o algoritmo DMB conseguiu com que o sistema atravessasse uma situação de sobrecarga de forma totalmente controlada e previsível, possibilitada pelo domínio do parâmetro *valor* de cada tarefa.

É válido lembrar que durante o uso de *valores* fixos foi ressaltada a importância do processo de escolha do *valor*, e essa importância cresce ainda mais com a sua modificação ao longo da execução. *Valores* desproporcionais podem transformar tarefas importantes em críticas e tarefas normais em descartáveis, diminuindo a serventia ou até mesmo prejudicando o comportamento esperado do sistema. Essa ressalva pode ser amenizada quando o uso de um componente monitor entra em cena, onde a atribuição subjetiva dos *valores* pode ser substituída pela imposição de restrições relativas.

Contudo, é preciso ressaltar a questão do custo computacional: se por um lado o uso de TUFs foi evitado anteriormente no intuito de não agregar um custo muito alto ao processo de escalonamento, esse custo pode estar presente na utilização de um componente monitor, devido à suas atribuições. O monitor deve literalmente definir o *valor* das tarefas em tempo de execução, baseando-se em resultados anteriores e nos requisitos impostos.

Uma alternativa apontada para o uso de um monitor é a definição prévia de modos de execução. Dessa forma o algoritmo de escalonamento torna-se novamente trivial, evitando o suposto *overhead* trazido com o monitoramento de resultados e definição *on-line* dos *valores*. Todavia, o uso de modos de execução também apresenta defeitos,

como a quantidade de comportamentos obtidos, que é limitada pelo número de modos do sistema (que devem sempre ser definidos previamente).

Baseando-se nos resultados apresentados nessa seção pode-se dizer que o objetivo de controle de tarefas em sistemas tempo real durante situações de sobrecarga foi atingido. Faz-se a ressalva, porém, do procedimento que será adotado para que esses resultados sejam observados, haja vista que cada um dos métodos apresentados aqui possui suas características positivas e negativas, e a situação onde eles serão utilizados deve influenciar muito na sua escolha.

Capítulo 6

Conclusões e Trabalhos Futuros

Ao longo do trabalho foram buscadas alternativas para superar as principais deficiências do escalonamento baseado em *deadlines*: subutilização do sistema e incapacidade de discernimento entre urgência e importância das tarefas em situações de sobrecarga. Além disso, buscou-se uma maneira de controlar o comportamento das tarefas (e conseqüentemente do sistema) durante esse tipo de situação.

O primeiro problema foi tratado através do uso da estratégia de escalonamento TAFT, que procura fugir de sobrecargas artificiais através do uso de tempos de execução esperados (ECET). Já o segundo problema foi abordado através do uso do parâmetro *valor* e de toda sua teoria, que conta com algoritmos escalonadores e métricas de desempenho próprios.

A idéia proposta por este trabalho visou unir as duas soluções citadas de forma a desfrutar dos benefícios trazidos por ambas. Apesar do relativo sucesso obtido por essa união, o almejado controle da execução das tarefas durante situações de sobrecarga não foi passível de ser feito utilizando-se dos algoritmos de escalonamento tradicionais. Assim, uma nova estratégia de escalonamento foi então desenvolvida com o uso do algoritmo DMB, proposto neste trabalho, que escalona tarefas baseando-se no parâmetro *valor* e na taxa de *deadlines* perdidos (ou EPs executadas, quando *TaskPairs* são usados).

Os resultados obtidos mostraram o DMB como uma solução bastante promissora no que se refere ao controle de falhas temporais, uma vez que seu desempenho avaliado objetivamente pelo critério de utilidade acumulada não esteja próximo do melhor possível. Através do uso de *valores* pré-definidos e fixos um comportamento altamente previsível da taxa de perdas foi apresentado entre as tarefas do sistema, demonstrando um alto grau de controle da execução.

Para avaliar a influência do uso de *valores* fixos nos resultados mostrados, uma nova análise foi realizada, dessa vez com mudanças durante o período simulado. Duas diferentes metodologias foram propostas: o uso de modos de execução e o uso de um

componente monitor. Enquanto a primeira estratégia permite definições prévias dos *valores* das tarefas e mudanças atômicas a qualquer instante, a segunda faz uso de restrições ligadas aos resultados para modificar o comportamento do sistema de forma gradual.

Procurando sempre garantir que o *overhead* apresentado pelo processo de escalonamento não tivesse um papel importante na medida dos resultados, optou-se por não empregar TUFs durante as simulações, embora elas façam parte da teoria do escalonamento através do *valor*. O *overhead* inerente dessa estratégia, atestado por trabalhos anteriores, não é condizente com a política de escalonamento TAFT. Da mesma forma são feitas restrições ao uso do componente monitor, que é capaz também de adicionar esse *overhead* à aplicação, dependendo da forma como for utilizado. O emprego de um processador auxiliar, responsável apenas pelo método de escalonamento, como sugerido por Locke em [15], muda totalmente o cenário e torna as ressalvas feitas desnecessárias.

Com o panorama de *valores* dinâmicos em cena, o DMB mostrou-se capaz de modificar o comportamento geral do sistema da forma desejada, através da mudança individual do *valor*. Uma política adequada de atribuição de *valores* torna possível o domínio da aplicação de forma bastante ampla, propiciando transformações na importância de tarefas (por exemplo, críticas em descartáveis e vice-versa) e até definição de grupos de serviços através desse parâmetro.

No entanto, é importante lembrar duas premissas básicas para a obtenção dos resultados esperados. A primeira diz respeito à escolha correta dos *valores*, que desempenha um papel fundamental durante a execução, pelo fato do algoritmo DMB ser altamente sensível a modificações desse parâmetro. *Valores* inadequados tornam o comportamento do DMB idêntico a algoritmos tradicionais, anulando os benefícios trazidos pelo seu uso.

A segunda premissa tem a ver com o uso conjunto do DMB com o escalonador TAFT, pelo fato do último ser responsável por evitar as perdas de *deadlines*. Embora funcione satisfatoriamente bem da maneira tradicional, o algoritmo DMB sofre muito com a sobrecarga adicional trazida pelo efeito dominó, e não é robusto o suficiente para reverter essa situação. Sob essas condições, transforma-se em uma solução inadequada a longo termo. *Valores* impróprios ou o uso do DMB da forma tradicional tornam o algoritmo uma solução que não apresenta ganho em relação a outras propostas.

Para a utilização em sistemas reais, a tendência não é achar ou criar um nicho próprio para a adição do algoritmo DMB sugerido, apenas implantá-lo em sistemas que se beneficiem da estratégia de escalonamento TAFT. Sua função seria substituir o parâmetro α-quantil na representação da importância da tarefa (ou seja, apenas durante o processo de escalonamento, sem modificações no uso desse parâmetro no monitoramento ou previsão do ECET). O parâmetro *valor* não tem muito sentido se aplicado em condições de *underload*, então o seu uso em aplicações que já desfrutam da

política TAFT de escalonamento (onde situações de sobrecarga, por menos freqüentes que sejam, são esperadas) seria bastante coerente.

Como sugestão para trabalhos futuros é apontada a implementação da estratégia proposta em um sistema real, tanto para comprovação dos resultados presentes na simulação como para observação dos fatores práticos, como *jitter*, tempo de troca de contexto e custo e complexidade computacionais do algoritmo.

Outro trabalho futuro a ser considerado é o uso do TAFT em um ambiente cujas tarefas apresentem funções de custo variável (*Variable Cost Functions* - VCF). A motivação é que a teoria matemática apresentada em trabalhos considerando esse tipo de função utiliza não somente o WCET, mas também o pior caso da função de custo da tarefa também. Dessa forma o pior caso de execução da tarefa é visto sob o pior caso em que ele pode ser executado, e garantias são baseadas nessa abordagem. Trata-se de um cenário um tanto quanto pessimista, que tende a apresentar uma grande subutilização do sistema, bastante apropriado para a implantação da estratégia TAFT em conjunto com o DMB. Sistemas que não necessitem 100% de garantia, ou seja, possam passar por sobrecargas transientes, podem se beneficiar muito das mudanças.

Referências Bibliográficas

[1] SCHEMMER, S. *A Middleware for Cooperating Mobile Embedded Systems*. Tese (Doctor's Thesis) — Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg, 2004.

[2] JENSEN, E.; LOCKE, C.; TOKUDA, H. A time driven scheduling model for real-time operating systems. In: *Proceedings IEEE Real-Time Sytems Symposium*. [s.n.], 1985. p. 112–122. Disponível em: <citeseer.ist.psu.edu/jensen85timedriven.html>.

[3] MCELHONE, C. G. *A Constrained Computational Model for Flexible Scheduling*. Tese (Doutorado) — University of York, 1996.

[4] TOKUDA, H.; WENDORF, J. W.; WANG, H.-Y. Implementation of a time-driven scheduler for real-time operating systems. In: *IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 1987. p. 271–280.

[5] WENDORF, J. W. Implementation and evaluation of a time-driven scheduling processor. In: *IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 1988. p. 172–180.

[6] BURNS, A. et al. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, v. 46, p. 305–325, 2000. Disponível em: <citeseer.ist.psu.edu/burns98meaning.html>.

[7] BECKER, L. B. et al. Robust scheduling in team-robotics. *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 77, n. 1, p. 3–16, 2005. ISSN 0164-1212.

[8] LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, ACM Press, New York, NY, USA, v. 20, n. 1, p. 46–61, 1973. ISSN 0004-5411.

[9] CHETTO, H.; CHETTO, M. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Software Eng.*, v. 15, n. 10, p. 1261–1269, 1989. Disponível em: <http://www.computer.org/tse/ts1989/e1261abs.htm>.

[10] PINEDO, M. *SCHEDULING - Theory, Algorithms, and Systems*. [S.l.]: Prentice Hall, 1995.

[11] LIU, J. W. S. W. *Real-Time Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN 0130996513.

[12] FARINES, J.-M.; FRAGA, J. d. S.; OLIVEIRA, R. S. de. *Sistemas de Tempo Real*. São Paulo-SP: IME-USP, 2000.

[13] SHA, L. et al. Real time scheduling theory: A historical perspective. *Real-Time Systems*, v. 28, n. 2-3, p. 101–155, 2004.

[14] GOUDA, M. G. et al. Distributed data processing technology, chapter 3, radar scheduling: Section 1, the scheduling problem. In: _____. [S.l.: s.n.], 1977. IV, cap. 3.

[15] LOCKE, C. D. *Best-effort decision-making for real-time scheduling*. Tese (Doutorado) — Carnegie Mellon University, 1986.

[16] CLARK, R. K. *Scheduling dependent real-time activities*. Tese (Doutorado) — Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1990. Disponível em: <citeseer.ist.psu.edu/clark90scheduling.html>.

[17] BUTTAZZO, G. C.; SPURI, M.; SENSINI, F. Value vs. deadline scheduling in overload conditions. In: *IEEE Real-Time Systems Symposium*. [s.n.], 1995. p. 90–99. Disponível em: <citeseer.ist.psu.edu/buttazzo95value.html>.

[18] DAVIS, R. et al. Flexible scheduling for adaptable real-time systems. In: *RTAS '95: Proceedings of the Real-Time Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 1995. p. 230. ISBN 0-8186-6980-2.

[19] LEHOCZKY, J.; THUEL, S. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive systems. *IEEE Real Time Symposium*, p. 110–123, December 1992.

[20] BONDAVALLI, A.; GIANDOMENICO, F. D.; MURA, I. Value-driven resource assignment in object-oriented real-time dependable systems. In: *Third International Workshop on Object-Oriented Real-Time Dependable Systems, Proceedings*. Newport Beach, CA, USA: [s.n.], 1997. p. 92–99.

[21] WU, H. et al. Cpu scheduling for statistically-assured real-time performance and improved energy efficiency. In: *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM Press, 2004. p. 110–115. ISBN 1-58113- 937-3.

[22] RAVINDRAN, B.; JENSEN, E. D.; LI, P. On recent advances in time/utility function real-time scheduling and resource management. In: *ISORC 2005. Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. [S.l.: s.n.], 2005. p. 55–60.

[23] STREICH, H. Taskpair-scheduling: An approach for dynamic real-time systems. In: *2nd Workshop on Parallel and Distributed Real-Time Systems Proceedings*. [S.l.: s.n.], 1994.

[24] STANKOVIC, J. A.; RAMAMRITHAM, K. The spring kernel: A new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, ACM Press, New York, NY, USA, v. 23, n. 3, p. 54–71, 1989. ISSN 0163-5980.

[25] BUTTAZZO, G.; STANKOVIC, J. *RED: Robust Earliest Deadline Scheduling*. 1993. Disponível em: <citeseer.csail.mit.edu/buttazzo93red.html>.

[26] NETT, E.; STREICH, H. The gmd-snake - real-time scheduling of a flexible robot application at run-time. In: *International Workshop on Parallel Computations and Scheduling*. [s.n.], 1997. Disponível em: <citeseer.ist.psu.edu/178201.html>.

[27] NETT, E.; GERGELEIT, M.; MOCK, M. Enhancing o-o middleware to become time-aware. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 20, n. 2, p. 211–228, 2001. ISSN 0922-6443.

[28] GERGELEIT, M. *A monitoring-based approach to object-oriented real-time computing*. Tese (Doutorado) — Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg, 2001.

[29] BECKER, L. B. *Um Método para Abordar todo o Ciclo de Desenvolvimento de Aplicações Tempo Real*. Tese (Doctor's Thesis) — PPGC - Universidade Federal do Rio Grande do Sul, 2003.

[30] BARABANOV, M. *A Linux-based Real-Time Operating System*. Dissertação (Mestrado) — New Mexico Institute of Mining and Technology, 1997. Disponível em: <citeseer.ist.psu.edu/barabanov97linuxbased.html>.

[31] DEAN, T.; BODDY, M. An analysis of time-dependent planning. In: *Proceedings of AAAI-88*. St. Paul, MN: [s.n.], 1988. p. 49–54.

[32] WEIDERMAN, N. Hartstone: Synthetic benchmark requirements for hard real-time applications. In: *Technical Report CMU/SEI-89-TR-023*. [S.l.: s.n.], 1989.

[33] OMNET++ User Manual. Disponível em: <http://www.omnetpp.org/doc/manual/usman.html>. Acesso em: jan 2006.

[34] MATSUMOTO, M.; NISHIMURA, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, v. 8, n. 1, p. 3–30, january 1998. ISSN 1049-3301.