Universidade Federal da Bahia

Universidade Salvador

Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Estratégias Baseadas em Servidores no Escalonamento de Sistemas de Tempo Real em Multiprocessadores

Ernesto de Souza Massa Neto

Programa Multiinstitucional de Pós-Graduação em Ciência da Computação – PMCC

Salvador - BA

2014

PMCC-DSc-0012

ERNESTO DE SOUZA MASSA NETO

ESTRATÉGIAS BASEADAS EM SERVIDORES NO ESCALONAMENTO DE SISTEMAS DE TEMPO REAL EM MULTIPROCESSADORES

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. George Marconi de Araújo Lima

Salvador
2014

Sistemas de Bibliotecas da UFBA

Massa Neto, Ernesto de Souza.
Estratégias baseadas em servidores no escalonamento de sistemas de tempo real em multiprocessadores / Ernesto de Souza Massa Neto. - 2014.
112f. : il.

Orientador: Prof. Dr. George Marconi de Araújo Lima.
Tese (doutorado) – Universidade Federal da Bahia, Instituto de Matemática, Universidade Estadual de Feira de Santana, Universidade Salvador, Salvador, 2014.

1. Processamento eletrônico de dados em tempo real. 2. Multicomputadores. 3. Sistemas de computação. 4. Cliente/Servidor (Computadores). I. Lima, George Marconi de Araújo. II. Universidade Federal da Bahia. Instituto de Matemática. III. Universidade Estadual de Feira de Santana. IV. Universidade Salvador. V. Título.

CDD - 004.33
CDU - 004.031.43

TERMO DE APROVAÇÃO

ERNESTO DE SOUZA MASSA NETO

ESTRATÉGIAS BASEADAS EM SERVIDORES NO ESCALONAMENTO DE SISTEMAS DE TEMPO REAL EM MULTIPROCESSADORES

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

Salvador, 10 de janeiro de 2014

PROFESSOR E ORIENTADOR GEORGE MARCONI LIMA, PH.D.
Universidade Federal da Bahia

PROFESSOR RAPHAEL PEREIRA DE OLIVEIRA GUERRA, DR.-ING.
Universidade Federal Fluminense

PROFESSOR OSMAR MARCHI DOS SANTOS, PH.D.
Universidade Federal de Santa Maria

PROFESSORA FABÍOLA GONÇALVES PEREIRA GREVE, PH.D.
Universidade Federal da Bahia

PROFESSOR FLÁVIO MORAIS DE ASSIS SILVA, DR.-ING.
Universidade Federal da Bahia

Para Pedro e Lyginha.
Sem eles, nada faria sentido.

AGRADECIMENTOS

Agradeço, em primeiro lugar, aos meus pais. Minha mãe, a minha mais fiel incentivadora. Uma conselheira que consegue enxergar muito além do que qualquer um de nós poderia. Meu pai, meu melhor e inesquecível amigo, me ensinou os valores que realmente importam. Eu não tomo nenhuma decisão sem que antes imagine o que vocês fariam em meu lugar.

Niquinha. Quis o destino que trilhássemos caminhos profissionais muito parecidos. Você desbravando e eu sempre seguindo os seus passos. Olhando para trás, foi seguindo você que eu cheguei até aqui.

Lyginha, minha esposa e amiga. Você decididamente é co-autora desta tese. Obrigado por estar sempre ao meu lado. Você e o Pedro enchem os meus dias de alegria. Professora Ednalva, minha querida sogra, obrigado por tê-los me dado.

Ao meu orientador George Marconi, um agradecimento especial. Além de me apresentar um mundo fascinante, você tornou possível que este trabalho fosse desenvolvido com extremo prazer. Nossas reuniões sempre resultam em boas ideias.

Agradeço ainda aos colegas Paul Regnier, José Augusto, Felipe Cerqueira e Greg Levin pelas colaborações no desenvolvimento deste trabalho e pelos simuladores dos algoritmos relacionados. Ao amigo Artur Kronbauer, obrigado pelo incentivo.

ABSTRACT

Real-time systems are characterized, not only by the need of being logically correct, but also by having to meet some timing requirements. The sequence in which tasks are executed is a relevant aspect in these systems and, if they are critical, temporal requirements are mandatory under risk of catastrophic consequences. The recent arising of multiprocessor systems has required all known scheduling strategies for uniprocessor, which are inefficient for multiprocessor environments, to be replaced by new strategies. The partitioned approach, which does not allow tasks to execute on more than one processor, and the global approach, which allows all tasks to execute on any processor, the two first developed approaches, usually present problems of efficiency or applicability. The semi-partitioned approach, a halfway point between the two others approaches, allocates tasks among processors, allowing just a few of them to execute on sets of processors. In order to avoid these tasks to be simultaneously scheduled on more than one processor, semi-partitioned algorithms define time reserves for these tasks, managing their execution. Taking advantage of servers, which guarantee temporal isolation between tasks that share a common environment, this work presents two new servers which provide time reservation and propose two new scheduling algorithms, one semi-partitioned and other global, for hard real-time systems in multiprocessor environments which use these servers. Experiments show that the performance of these algorithms are comparable to the main related scheduling algorithms and furthermore one of them was proven to be optimal with respect to the use of processing capacity on multiple processors.

Keywords: Real-Time Systems, Multiprocessor, Scheduling, Optimality, Server

RESUMO

Sistemas de tempo real se caracterizam não somente pela necessidade do seu correto processamento lógico, mas também por terem que atender requisitos temporais, sendo a ordem escolhida para a execução das suas tarefas um aspecto particularmente relevante. Quando estes sistemas são classificados como críticos, o atendimento dos requisitos temporais é obrigatório, sob a pena de consequências catastróficas. A recente proliferação de sistemas computacionais compostos por múltiplos processadores fez com que as estratégias de escalonamento consolidadas para sistemas com uma única unidade de processamento, porém ineficientes para ambientes com múltiplos processadores, tivessem que ser substituídas por novas estratégias. As abordagens particionada, que não permite a qualquer tarefa executar em mais de um processador, e global, que permite a todas as tarefas executem em qualquer processador, inicialmente adotadas, usualmente apresentam problemas de eficiência ou de aplicabilidade, abrindo espaço para a abordagem semi-particionada. Esta estratégia distribui as tarefas pelos processadores, permitindo que apenas algumas tarefas específicas tenham permissão pera executar em dois ou mais processadores. Para que nenhuma destas tarefas específicas execute simultaneamente em mais de um processador, os algoritmos que seguem abordagens semi-particionadas definem reservas de tempo em que estas ocuparão os processadores, controlando a sua execução. Aproveitando-se de servidores, que garantem isolamento temporal entre tarefas que executam em um mesmo ambiente, este trabalho apresenta dois novos servidores como instrumento para implementar reservas de tempo, propondo dois novos algoritmos de escalonamento, um semi-particionado e o outro global, para sistemas de tempo real críticos em ambientes com múltiplos processadores que se utilizam destes servidores. Os experimentos realizados revelaram que os desempenhos destes algoritmos são comparáveis aos principais algoritmos de escalonamento relacionados, sendo que um dos algoritmos desenvolvidos foi provado ser ótimo com relação ao uso da capacidade computacional em múltiplos processadores.

Palavras-chave: Sistemas de Tempo Real, Multiprocessador, Escalonamento, Otimalidade, Servidor

SUMÁRIO

LISTA DE FIGURAS

Capítulo 1

The simple fact that a task can use only one processor
even when several processors are free at the same time
adds a surprising amount of difficulty to the scheduling of multiple processors.

Liu(1969)

INTRODUÇÃO

Sistemas computacionais que se caracterizam não somente pela necessidade do seu correto processamento lógico, mas também por terem de atender certos requisitos temporais, são denominados por sistemas de tempo real [Ramamrithan e Stankovic 1994]. Um aspecto particularmente relevante para esta classe de sistemas computacionais é a ordem escolhida para a execução de cada uma das suas tarefas [Codd 1960], sendo que a escolha de um sequenciamento adequado está estreitamente ligada ao atendimento das especificações temporais definidas em seus requisitos.

A crescente miniaturização dos dispositivos eletrônicos, aliada ao avanço da tecnologia de redes de comunicação, fez com que os sistemas de tempo real se tornassem um componente cada vez mais presente no cotidiano. Eles vêm assumindo, de forma pervasiva, novas tarefas que vão desde o inofensivo controle de transmissões multimídia até o crítico controle dos freios em automóveis, chegando ao monitoramento e controle de complexas operações em uma planta industrial. Estes são bons exemplos da proliferação dos sistemas de tempo real e servem de prenúncio para novas aplicações que proporcionarão segurança e qualidade de serviços.

Quando os ambientes computacionais possuem múltiplas unidades de processamento, passa a existir a possibilidade de execução simultânea de tarefas e as estratégias de sequenciamento aplicadas precisam ser distintas. O crescente uso de sistemas com esta característica aumenta a relevância deste problema e motiva a comunidade de tempo real a buscar o desenvolvimento de mecanismos de escalonamento eficientes para multiprocessadores.

1.1 SISTEMAS DE TEMPO REAL

Pode-se entender um sistema de tempo real como parte de um sistema mais abrangente que pode ser decomposto em três subsistemas menores. O primeiro subsistema é constituído por

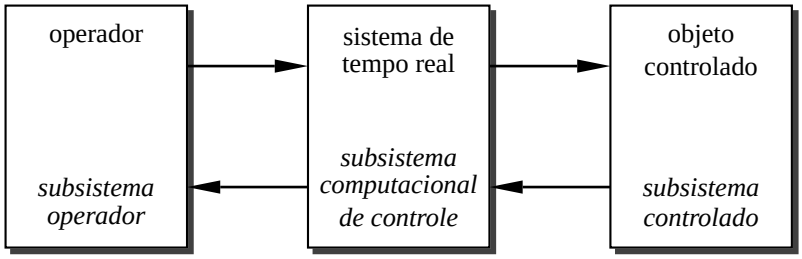


Figura 1.1. Modelo de Sistema de Tempo Real [Kopetz 2011]

uma planta física ou objeto a ser controlado, o segundo subsistema é o sistema de tempo real encarregado do controle deste objeto, e o terceiro subsistema é um operador humano [Kopetz 2011] (Figura 1.1). A interface entre o operador humano e o sistema de tempo real é chamada de interface homem-máquina, enquanto a interface entre o sistema de tempo real e o objeto controlado é chamada de interface de instrumentação. Enquanto a primeira interface é composta por dispositivos de entrada/saída, a segunda consiste especificamente em sensores e atuadores que respectivamente capturam informações do ambiente e enviam sinais de controle ao objeto controlado.

As tarefas que compõem um sistema de tempo real geralmente são executadas de forma recorrente, sendo a sua repetição potencialmente infinita. Cada execução de uma tarefa é chamada de *job* e é composta por conjuntos de instruções que podem, por exemplo, estarem associadas ao processamento dos dados capturados por sensores e que são disponibilizados em intervalos fixos de tempo. Neste caso, o período de tempo decorrido entre duas liberações sucessivas de *jobs* de uma mesma tarefa é constante, sendo estas tarefas classificadas como periódicas [Liu e Layland 1973]. Um conjunto de tarefas periódicas pode ainda ser classificado como síncrono, no caso de todas as primeiras liberações ocorrerem simultaneamente, ou assíncrono caso contrário. Quando o período de tempo decorrido entre duas liberações sucessivas de *jobs* de uma mesma tarefa precisar atender a um intervalo mínimo, esta tarefa é classificada como esporádica [Mok 1983]. Tarefas associadas com eventos randômicos, geralmente externos ao sistema e que podem se repetir ou não, são classificadas como aperiódicas. Por estarem sujeitas a rajadas de liberações de *jobs*, não é possível fornecer quaisquer garantias temporais a esta classe de tarefas e, como resultado, elas não podem ser tarefas críticas [Burns 1991].

Sistemas de tempo real podem ser classificados como críticos ou não críticos. Os sistemas de tempo real não críticos têm entre os seus requisitos a produção de uma resposta em um tempo "razoável", ou seja, é desejável que eles concluam a sua execução dentro de um certo prazo (*deadline*), porém existe uma pequena tolerância ao não atendimento deste requisito temporal. Por outro lado, sistemas de tempo real críticos devem atender rigorosamente aos seus *deadlines*, sendo considerados incorretos caso contrário [Manacher 1967]. Algumas vezes os sistemas críticos são caracterizados pelo não atendimento de um *deadline* ser potencialmente catastrófico, enquanto que o não atendimento de um *deadline* em um sistema não crítico apenas acarretaria

uma redução na qualidade do serviço fornecido [Burns 1991]. Sistemas multimídia são bons exemplos de sistemas de tempo real não críticos, enquanto que os sistemas controladores dos freios ABS e os sistemas transponders dos aviões são casos de sistemas de tempo real críticos.

Com base no que foi exposto até aqui, pode-se definir formalmente alguns importantes conceitos pertinentes às tarefas de tempo real:

Definição 1.1.1 (Job)**.** *Um job $J{:}(c, r, d)$ liberado por uma tarefa τ, ou uma instância de τ, é uma sequência de instruções executadas de forma sequencial por τ que consome até c unidades de tempo de processamento e deve iniciar e terminar a sua execução no intervalo $[r, d)$. Ou seja, c, r, e d representam respectivamente o tempo gasto no pior caso pela instância J de τ em um processador durante a sua execução, o instante em que J foi liberado para execução, e o deadline absoluto de J que consiste no prazo máximo para que J conclua a sua computação.*

Definição 1.1.2 (Período)**.** *Período é o tempo decorrido entre duas liberações sucessivas de jobs de uma mesma tarefa periódica ou o mínimo tempo decorrido entre duas liberações sucessivas de jobs de uma mesma tarefa esporádica.*

Definição 1.1.3 (*Deadline* Relativo)**.** *Quando a diferença entre o deadline absoluto de todos os jobs de uma mesma tarefa e as suas respectivas liberações for um valor constante, este será o deadline relativo desta tarefa.*

De acordo com as Definições 1.1.1 e 1.1.3, pode-se observar que um *deadline* relativo é atributo de uma tarefa, enquanto *deadline* absoluto é um atributo de um *job*. Portanto, sempre que for feita referência ao *deadline* de uma tarefa, o referido será um *deadline* relativo. Por outro lado, quando for feita referência ao *deadline* de um *job*, este será um *deadline* absoluto.

A comparação entre os *deadlines* relativos e os períodos de um conjunto de tarefas produz uma classificação adicional. Quando o *deadline* de uma tarefa é definido pelo seu próprio período, esta tarefa é definida como tendo *deadline* implícito. No caso em que os *deadlines* sempre são limitados superiormente pelo período das tarefas, diz-se que estas possuem *deadline* restrito. No caso derradeiro, quando não existe uma relação entre os *deadlines* e os períodos, as tarefas são classificadas como tendo *deadline* arbitrário [Goossens et al. 2004]. Mais uma vez é possível notar que tarefas com *deadline* implícito são casos particulares de tarefas com *deadline* restrito, e estas por sua vez são casos particulares de tarefas com *deadline* arbitrário.

Neste trabalho, uma tarefa de tempo real será considerada correta se ela concluir totalmente a sua computação até o seu *deadline*. No entanto, o conhecimento *a priori* do custo computacional exato da execução de uma tarefa é algo improvável de ser obtido. Estruturas de linguagens de programação, como laços condicionais, impossibilitam que seja conhecido com antecedência o tempo exato em que uma tarefa irá ocupar um processador durante a sua execução. Além disso, alguns sistemas permitem que a execução de tarefas sejam suspensas para que sejam executadas tarefas mais prioritárias. Estas preempções provocam um gasto adicional de tempo com trocas de contexto, sendo a determinação precisa destes custos ainda objeto de estudo com

resultados recentes sendo encontrados em Paolieri et al.(2009). A perfeita sincronização entre a disponibilização de dados pelos sensores, a execução das tarefas de tempo real e o início do trabalho dos atuadores é impossível de ser obtido na prática. Isso introduz um consumo adicional de tempo na interface de instrumentação, chamado de *jitter* [Marti et al. 2001]. Fatores como estes, unidos aos fatores associados ao *hardware* como o uso da memória cache, fazem com que o custo computacional estimado para uma tarefa não reflita o tempo exato gasto na execução das suas instruções, mas seja um limite superior a todo o tempo em que um processador ficará ocupado com trabalhos relacionadas a essa respectiva tarefa, ou o seu tempo de execução no pior caso ("*worst case execution time*"). Com isso pode ser formalmente definida mais uma característica para uma tarefa de tempo real (1.1.4).

Definição 1.1.4 (Custo de Execução)**.** *O custo de execução de uma tarefa é o limite superior para todo custo computacional provocado pela execução de cada um dos seus jobs.*

O compartilhamento de outros recursos além do processador propicia uma possível interferência entre as tarefas de um sistema. Neste caso, uma tarefa pode sofrer interferência não apenas de tarefas mais prioritárias, mas também de tarefas menos prioritárias que já estejam alocando, de forma exclusiva, recursos necessários a sua execução. A ocorrência de tarefas que possuam entre si uma relação de precedência insere, ainda, a necessidade de concluir a execução de uma tarefa antecessora antes da ativação da sua sucessora de forma a garantir um correto resultado no processamento do sistema. A existência desta relação interfere na decisão do sequenciamento das tarefas, devendo as tarefas precedentes receber maior prioridade para serem executadas. A existência de dependência entre tarefas e o compartilhamento de outros recursos além dos processadores já justificariam um estudo independente, como em Audsley et al.(1993).

O modelo utilizado neste trabalho corresponde a um sistema ideal, considerando processadores idênticos, tarefas críticas, preemptíveis, independentes e sem o compartilhamento de recursos. Assim, qualquer tarefa pode sofrer preempções a qualquer momento durante a sua execução e, quando interrompida em um processador, pode retomar a sua execução em qualquer outro processador do sistema que esteja disponível.

Custos com trocas e transferências de contexto entre processadores podem ocorrer diversas vezes para um mesmo *job*, sendo importante que se conheça os limites superiores para as quantidades de preempções e de migrações de *jobs* entre processadores. Nas abordagens apresentadas nos capítulos seguintes, todos os custos computacionais decorrentes da execução de uma tarefa são considerados como já inclusos no seu custo de execução, não sendo considerados de forma particular.

Para que um sistema tenha suas tarefas concluídas até um tempo desejado, é preciso que as operações efetuadas por todas as tarefas do sistema sejam sequenciadas de forma apropriada. O estudo de quais tarefas devem ser executadas prioritariamente e a criticidade do atendimento às suas restrições temporais possuem significativa importância para um bom sequenciamento.

Várias políticas baseadas em prioridades já foram propostas, utilizando parâmetros como o prazo máximo para a conclusão da execução, ou a tarefa executada mais frequentemente. A determinação da existência de um sequenciamento de tarefas que atenda a todos os requisitos temporais de um sistema e a determinação de que, caso esta sequência exista, um dado algoritmo sempre conseguirá encontrá-la são conhecidos como problemas de escalonamento em sistemas de tempo real.

1.2 O PROBLEMA DE ESCALONAMENTO

O problema de escalonar tarefas consiste no estabelecimento de regras que garantam o atendimento dos requisitos temporais especificados para um sistema de tarefas. Este estudo envolve o sequenciamento adequado do uso dos recursos compartilhados, de forma que o comportamento temporal do sistema seja compreensível e previsível [Stankovic 1988]. Este sequenciamento é produzido por um escalonador, que pode ser entendido como um algoritmo que associa, ao longo do tempo, processadores aos *jobs* liberados pelo sistema de tarefas. Para que um escalonamento seja *válido*, é preciso que em qualquer instante t um processador execute no máximo um *job*; o mesmo *job* não pode ser executado simultaneamente em mais de um processador; nenhum *job* execute mais de uma vez; e cada *job* execute por até c unidades de tempo até o seu *deadline*. Um sistema de tarefas que permite ao menos um escalonamento válido é chamado de sistema *factível* [Joseph e Goswami 1989].

Dois tipos de problemas podem ser identificados no estudo do escalonamento de sistemas críticos em tempo real: o problema de estabelecer um sequenciamento na ordem de execução de um conjunto de tarefas e o problema de analisar se um sistema de tarefas é escalonável sob um dado algoritmo de escalonamento. O primeiro problema visa encontrar um algoritmo de escalonamento que, em tempo de execução, atenda a todos os *deadlines* de um sistema de tarefas que se saiba previamente factível [Goossens et al. 2004]. Já a análise da escalonabilidade busca predizer o comportamento temporal de um sistema de tarefas sob dado algoritmo de escalonamento através de testes que determinam se as suas restrições temporais serão atendidas, ou não, em tempo de execução. Esta análise inclui diversos fatores como as características do modelo utilizado e a realização de testes de escalonabilidade [Sha et al. 2004]. O ideal seria a criação de condições suficientes e necessárias para a escalonabilidade, mas este é um problema fortemente co-NP-completo para sistemas arbitrários, sendo tratável apenas em alguns casos particulares [Baruah et al. 1993]. Um critério comumente utilizado na análise da escalonabilidade para sistemas de tempo real é o limite superior (*upper bound*) para taxa de utilização permitida aos processadores, abaixo do qual é garantido para qualquer sistema que todos os *deadlines* serão atendidos por um dado algoritmo [Stankovic 1988].

Durante décadas, pesquisadores envolvidos com sistemas de tempo real concentraram-se no escalonamento em um único processador, já tendo estudado largamente este problema. Este contexto produziu importantes resultados e este problema já se encontra razoavelmente resol-

vido. No entanto, nos últimos anos, o escalonamento em multiprocessadores vem ganhando destaque devido ao surgimento de novas arquiteturas que utilizam mais de um processador e devido à melhoria na qualidade nas redes de comunicação. Máquinas *multicore* invadiram o mercado, propiciando um ambiente favorável à distribuição de processos por múltiplos processadores. Desktops de baixo custo equipados com processadores de até oito núcleos já podem ser facilmente encontrados nas lojas, e essa tendência parece que vai se estender por algum tempo pois fabricantes oferecem chips de alto desempenho com até 72 núcleos [Tilera 2013], além de existirem projetos com ainda mais unidades de processamento [Intel 2012].

Dentro desta nova realidade, quando um sistema de tarefas deve ser executado em um ambiente com múltiplos processadores, é necessário que seja tomada a decisão de quando cada tarefa deverá ocupar algum dos processadores e quando deverá aguardar para executar em um instante futuro. O conjunto de regras responsável por esta decisão deve, entre outras coisas, decidir quais tarefas devem executar concorrentemente [Codd 1960].

As pesquisas sobre o escalonamento em multiprocessadores ainda estão em seu início se comparadas às pesquisas sobre escalonamento em um único processador. Esta situação se agrava ao se constatar que a maioria dos resultados conhecidos para sistemas uniprocessados não podem ser facilmente aplicados aos sistemas multiprocessados [Andersson e Jonsson 2002]. Este cenário motiva a comunidade de tempo real a buscar o desenvolvimento de mecanismos de escalonamento eficientes para multiprocessadores, e bons resultados têm sido obtidos nos últimos anos.

O problema abordado neste trabalho delimita-se pelo escalonamento de um conjunto Γ composto por n tarefas esporádicas independentes e preemptíveis em um conjunto Π composto por m processadores idênticos. Cada tarefa $\tau \in \Gamma$ libera uma sequência (possivelmente infinita) de *jobs* a serem executados em processadores de Π, sendo proibida a execução simultânea de um mesmo *job* em dois processadores diferentes. Portanto, para que um conjunto de tarefas Γ seja escalonado corretamente, cada um dos *jobs* $J:(c,r,d)$ liberados pelas tarefas em Γ deve iniciar a sua execução após a sua liberação no instante r, executar em um ou mais processadores (nunca simultaneamente) por um tempo máximo igual a c, e finalizar a sua execução antes do instante d.

A tese defendida neste trabalho apresenta novas estratégias baseadas em reserva de banda para o escalonamento de tarefas de tempo real em arquiteturas com múltiplos processadores. O controle das migrações é implementado de forma a garantir a escalonabilidade do sistema com a aplicação do algoritmo EDF (*Earliest Deadline First*) [Liu e Layland 1973], ótimo com relação a utilização de processadores para sistemas com um único processador. Seguindo a convenção usual, o custo com preempções e migrações entre processadores não é levado em consideração, apesar da quantidade de ocorrência destes eventos ser utilizada como fator de comparação entre algoritmos de escalonamento.

As estratégias de escalonamento propostas foram batizadas como EDF-BR (*Earliest Deadline First with Bandwidth Reservation*) e QPS (*Quasi-Partitioning Scheduling*), sendo a pri-

meira caracterizada pela utilização de servidores para evitar que uma mesma tarefa seja escalonada simultaneamente em processadores distintos enquanto a segunda utiliza servidores para garantir que, em intervalos de tempo específicos, tarefas pertencentes a grupos distintos executem simultaneamente em processadores diferentes. Ambas as estratégias foram avaliadas através de simulações e mostraram-se competitivas quando comparadas com trabalhos relacionados.

O EDF-BR pertence a classe das abordagens semi-particionadas que escalonam tarefas esporádicas em múltiplos processadores. As tarefas são distribuídas sequencialmente pelos processadores e, quando não for mais possível a alocação total de nenhuma tarefa no processador atual, é escolhida uma tarefa para migrar entre este processador e o próximo. Assim como outros algoritmos semi-particionados [Andersson e Bletsas 2008] [Andersson e Tovar 2006] [Bletsas e Andersson 2009], EDF-BR divide o fluxo do tempo em janelas. As tarefas que migram evitam ocupar dois processadores ao mesmo tempo dividindo a sua execução proporcionalmente pelas janelas de tempo, executando em um processador no início e em outro processador ao final de cada janela. O desempenho do EDF-BR é comparável aos principais algoritmos relacionados. Em particular, por seu algoritmo executar em tempo polinomial, EDF-BR é adequado para o escalonamento de sistemas de tarefas complexos que executam em arquiteturas com múltiplos processadores.

O algoritmo QPS é um algoritmo ótimo no sentido em que escalona corretamente qualquer sistema de tarefas esporádicas com *deadline* implícito que tenha utilização total não superior à quantidade de processadores disponíveis. QPS apresenta uma nova forma de se particionar um conjunto de tarefas, denominada por quasi-particionamento. Uma das principais características do quasi-particionamento é a propriedade de que, ao ser removida uma tarefa qualquer de uma quasi-partição, o conjunto de tarefas restante nesta quasi-partição é escalonável por EDF em um único processador. Esta propriedade permite ao QPS adotar uma estratégia adaptativa interessante. Enquanto as tarefas chegam periodicamente, QPS controla a sua execução com um conjunto de servidores. Ao ocorrer atraso na chegada de alguma tarefa esporádica, QPS altera o seu comportamento para adotar um escalonamento particionado. Os experimentos realizados mostram que QPS reduz consideravelmente a quantidade de migrações e de preempções quando comparado com trabalhos relacionados.

Além da introdução, apresentada neste capítulo, este trabalho encontra-se estruturado da seguinte forma: o Capítulo 2 apresenta o modelo de sistema adotado, as notações utilizadas e faz uma breve retomada dos principais resultados relacionados na área de escalonamento em sistemas em tempo real. O Capítulo 3 apresenta dois novos tipos de servidores, o servidor de Bloqueio Regular(BR) e o servidor de Taxa-Fixa, que serão utilizados pelas estratégias de escalonamento EDF *with Bandwidth Reservation* (EDF-BR) e Escalonamento Quasi-Particionado (QPS), apresentadas nos Capítulos 4 e 5, respectivamente. O sexto e último capítulo é destinado às conclusões.

Capítulo 2

Uma viva inteligência de nada serve
se não estiver ao serviço de um caráter justo;
um relógio não é perfeito quando trabalha rápido,
mas sim quando trabalha certo.

Luc de Clapiers, marquês de Vauvenargues

I don't need time,
I need a deadline

Duke Ellington

ALGORITMOS DE ESCALONAMENTO EM SISTEMAS DE TEMPO-REAL

Entre os aspectos que podem ser considerados por um algoritmo de escalonamento encontra-se a urgência na execução de cada tarefa [Codd 1960]. Este aspecto se apresenta particularmente importante para sistemas de tempo real, para os quais os requisitos temporais são relevantes ao ser verificada a sua correção. Um sistema é correto se atende a um conjunto pré-determinado de especificações e, para sistemas de tempo real, isso envolve o compartilhamento de recursos no tempo, como a quantidade de processadores disponíveis e o tamanho da memória compartilhada [Joseph e Goswami 1989].

Neste trabalho é considerado um modelo de tarefas ideal em que todas as tarefas são críticas, preemptíveis, independentes e sem o compartilhamento de outros recursos além dos processadores. Desta forma, a qualquer momento a execução de uma tarefa pode ser suspensa em um processador e ser retomada posteriormente em um outro processador do sistema, sendo que os custos com as trocas de contexto e as transferências de contexto entre processadores já estão incluídos nos custos de execução das tarefas. Apesar dos custos com migrações e preempções não serem mensurados explicitamente, alguns algoritmos apresentados têm seus desempenhos avaliados pela frequência com que estes eventos ocorrem. Embora isto não represente fielmente a realidade, esta é uma simplificação usualmente adotada no tratamento do escalonamento de sistemas.

2.1 NOTAÇÕES E MODELO UTILIZADO

Embora existam diversos modelos de tarefas descritos na literatura [Baruah et al. 1999, Fisher 2007], a definição de tarefa utilizada neste trabalho é um pouco mais geral do que a usualmente encontrada. Isto deve-se à necessidade de representar agregações de tarefas no Capítulo 5.

Uma tarefa τ é caracterizada pela sua taxa de execução, denotada por $\mathrm{R}(\tau)$, que indica a fração do tempo em que ela requer o uso de um processador para a execução das suas instruções. Quando uma tarefa τ libera um job $J{:}(c, r, d)$, seu tempo de execução c é igual a $\mathrm{R}(\tau)(d - r)$. Isto é, c não é sempre o mesmo para uma tarefa τ, mas depende explicitamente da extensão do intervalo $[r, d)$ em que um job J de τ será executado. Em todo caso, tem-se que a relação 2.1 será sempre verdadeira para todo job $J_i{:}(c_i, r_i, d_i)$ de τ.

$$\mathrm{R}(\tau) = \frac{c_i}{d_i - r_i} \tag{2.1}$$

Note que o modelo de tarefa frequentemente utilizado considera $d - r$ constante e, portanto, constitui-se em um caso particular do modelo adotado neste trabalho.

Para uma tarefa τ em um instante t, denota-se por $D(\tau, t)$ o seu próximo *deadline* e por $E(\tau, t)$ o tempo de execução necessário para concluir o job corrente. Também é assumido que uma tarefa nunca poderá liberar um novo job antes do *deadline* do seu job corrente. Quando tratamos com um conjunto de tarefas Γ, são utilizadas as notações $\mathrm{R}(\Gamma)$ e $D(\Gamma, t)$ para representar respectivamente o somatório das taxas de execução das tarefas em Γ e o próximo *deadline* dos seus jobs ativos após t. Formalmente, $\mathrm{R}(\Gamma) = \sum_{\tau_i \in \Gamma} \mathrm{R}(\tau_i)$ e $D(\Gamma, t) = \min_{\tau_i \in \Gamma}\{D(\tau_i, t)\}$.

Estes conceitos são formalizados através das Definições 2.1.1, 2.1.2 e 2.1.3.

Definição 2.1.1 (Tarefa de Taxa-Fixa)**.** *Uma* tarefa de taxa-fixa*, ou simplesmente* tarefa, $\tau : \mathrm{R}(\tau)$ *libera uma sequência possivelmente infinita de jobs* $J_i{:}(c_i, r_i, d_i)$ *com taxa de execução* $\mathrm{R}(\tau) = c_i/d_i - r_i, \forall J_i$ *de* τ*.*

Portanto, um job $J{:}(c, r, d)$ de τ liberado no instante r deverá ocupar processadores por um total de $c = E(\tau, r) = \mathrm{R}(\tau)(d - r)$ unidades de tempo até o instante $d = D(\tau, r)$.

Definição 2.1.2 (Tarefa com *Deadline* Restrito)**.** *Uma tarefa* $\tau{:}(C, D, T)$ *com deadline restrito é um caso particular de tarefa em que* T *é o intervalo mínimo entre a liberação de dois jobs consecutivos,* $C = \mathrm{R}(\tau)T$ *e* $D = (d - r) \leq T, \forall$ *job* $J{:}(c, r, d)$ *de* τ*, é o seu deadline relativo.*

Definição 2.1.3 (Tarefa com *Deadline* Implícito)**.** *Quando ocorrer de uma tarefa* τ *possuir* $D = T$*, ela será classificada como tarefa de deadline implícito e a sua notação poderá ser simplificada para* $\tau{:}(C, T)$*.*

Outros dois conceitos importantes utilizados pelas estratégias de escalonamento apresentadas neste trabalho são os conceitos de tarefas e de jobs ativos, apresentados nas Definições 2.1.4 e 2.1.5.

Definição 2.1.4 (Job Ativo)**.** *Um job $J{:}(c, r, d)$ é considerado ativo durante o intervalo $[r, d)$ e inativo em todos os demais instantes.*

Definição 2.1.5 (Tarefa Ativa)**.** *Uma tarefa estará ativa quando possuir um job ativo, caso contrário ela estará inativa.*

É importante observar que um job, mesmo que já tenha completado toda a sua execução, permanecerá ativo até a chegada do seu deadline. Esse conceito é fundamental para a compreensão do algoritmo apresentado no Capítulo 5.

2.2 ALGORITMOS DE ESCALONAMENTO PARA SISTEMAS UNIPROCESSADOS

Quando a atribuição de prioridade às tarefas do sistema é feita de forma fixa, todos os jobs de uma mesma tarefa possuem a mesma prioridade e o problema de escalonamento é reduzido à definição da prioridade de cada tarefa, sendo sempre executado o job da tarefa mais prioritária. Já nos algoritmos de escalonamento com prioridade dinâmica, a prioridade atribuída a uma tarefa, e até mesmo a um job de uma tarefa, pode ser alterada com o tempo [Goossens et al. 2004].

2.2.1 Algoritmos Elementares

Em plataformas com um único processador, o algoritmo *Rate Monotonic* [Liu e Layland 1973] mostrou-se ótimo entre os algoritmos com prioridade fixa para sistemas com tarefas periódicas com *deadline* implícito para o caso em que as tarefas são síncronas, ou seja, todas as tarefas liberam o seu primeiro job simultaneamente [Liu e Layland 1973, Goossens e Devillers 1997]. Isto significa que se algum dos algoritmos de escalonamento desta classe conseguir um escalonamento válido para um determinado sistema de tarefas, o *Rate Monotonic* também produzirá um escalonamento válido para este mesmo sistema. O *Rate Monotonic* atribui prioridades em proporção inversa ao período das tarefas e garante escalonar qualquer conjunto Γ com n tarefas que possuam utilização total que atenda a inequação 2.2.

$$\sum_{\tau_i \in \Gamma} \mathrm{R}(\tau_i) \leq n(\sqrt[n]{2} - 1) \tag{2.2}$$

Note que este limite tende a $ln(2)$ para uma quantidade infinita de tarefas. Como o *Rate Monotonic* é ótimo, tem-se que o mínimo limite superior para n grande é de $ln(2) \approx 0.693$ quando se deseja garantir que um sistema de tarefas periódicas síncronas com *deadline* implícito seja escalonado corretamente por um algoritmo de escalonamento com prioridade fixa. No caso em que as tarefas periódicas possuem *deadline* restrito, o algoritmo *Deadline Monotonic* [Leung e Whitehead 1982], que atribui prioridade fixa para as tarefas de forma inversa ao seu deadline, foi provado ser ótimo por produzir escalonamento válido para qualquer sistema de tarefas com *deadline* restrito que seja escalonável por algum outro algoritmo com prioridade fixa.

Entre os algoritmos com prioridade dinâmica, Liu e Layland(1973) propuseram o algoritmo EDF (*Earliest Deadline First*), que atribui aos jobs uma prioridade inversa ao seu *deadline* absoluto. O EDF é um algoritmo ótimo para o escalonamento de tarefas periódicas com *deadline* implícito [Liu e Layland 1973], sendo capaz de escalonar corretamente qualquer conjunto Γ com n tarefas desde que seja satisfeita a condição expressa pela inequação 2.3.

$$\sum_{\tau_i \in \Gamma} \mathrm{R}(\tau_i) \leq 1 \tag{2.3}$$

Satisfazer esta condição não é apenas suficiente, mas também é necessário para que um sistema de tarefas periódicas com *deadline* implícito seja escalonável por EDF. Este é um dos casos particulares mencionados na Seção 2.2 em que existe um teste de escalonabilidade suficiente e necessário. Para sistemas com tarefas não periódicas, o EDF também é um algoritmo ótimo no sentido em que se algum algoritmo de escalonamento produzir um escalonamento válido para um dado sistema de tarefas, EDF também será capaz de fazê-lo para o mesmo sistema de tarefas [Dertouzos 1974].

Na verificação da escalonabilidade para EDF em sistemas com tarefas de *deadline* restrito, os testes geralmente envolvem a análise do limite para a demanda gerada nos processadores [Burns et al. 2012]. Estes testes podem ser executados em tempo pseudo-polinomial para a maioria dos sistemas e foram derivados para quando as tarefas forem periódicas [Baruah et al. 1990a] e para quando elas forem esporádicas [Baruah et al. 1990b]. Segundo Burns et al.(2012) e Sha et al.(2004), estes testes consistem basicamente em verificar, para todo intervalo $[t_1, t_2)$ onde t_1 e t_2 são *deadlines* de alguma tarefa e $t_2 - t_1$ é limitado por um certo valor L, se a soma da demanda de todos os jobs liberados por um conjunto de tarefas Γ a partir de t_1 e com *deadline* anterior a t_2 é inferior a $t_2 - t_1$. Esta verificação foi provada por Baruah et al.(1993) ser equivalente à verificação no intervalo $[0, t_2 - t_1)$ e é formalmente representada pela equação 2.4

$$h(t) = \sum_{\forall \tau_i \in \Gamma} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \mathrm{R}(\tau_i) T_i \leq t \tag{2.4}$$

Inicialmente foi adotado para L o valor do MMC (Mínimo Múltiplo Comum) dos períodos das tarefas, denominado por hiperperíodo de Γ. Outros limites mais estreitos foram obtidos posteriormente, como os derivados por Spuri e Buttazzo(1996), apresentado pela Equação 2.5.

$$s^{q+1} = \sum_{\forall \tau_i \in \Gamma} \left\lceil \frac{s^q}{T_i} \right\rceil \mathrm{R}(\tau_i) T_i \tag{2.5}$$

Esta equação iterativa termina quando $s^{q+1} = s^q$, tendo $s^0 = \sum_{\forall \tau_i \in \mathcal{T}} \mathrm{R}(\tau_i) T_i$, e então temos $L = s^q$. Quando a utilização total do sistema for estritamente inferior a 1, Hoang et al.(2006) mostram que L pode ser calculado de forma mais simples através da Equação 2.6.

$$L = \max \left\{ D_1, D_2, \ldots, D_n, \frac{\sum_{j=1}^{n} (T_j - D_j)\,\mathrm{R}(\tau_j)}{1 - \mathrm{R}(\Gamma)} \right\} \tag{2.6}$$

Um aperfeiçoamento do teste exato de demanda, denominado QPA (*Quick convergence Processor-demand Analysis*), foi apresentado recentemente por Zhang e Burns(2009) e reduziu os pontos em que a demanda total do sistema deve ser avaliada. Neste teste, L é calculado inicialmente como o mínimo valor entre os obtidos pelas Equações 2.5 e 2.6. Note que, quando $\sum_{\forall \tau_i \in \Gamma} \mathrm{R}(\tau_i) = 1$, apenas a Equação 2.5 pode ser utilizada. QPA adota t como o maior *deadline* absoluto das tarefas em Γ não superior a L, e utiliza t como o primeiro instante a ser avaliado através do teste $h(t) \leq t$ (Equação 2.4). Em caso de sucesso, faz $t = h(t)$ e reaplica este mesmo teste recursivamente até que t seja inferior ao menor *deadline* relativo das tarefas em Γ. Apesar de não ser um algoritmo polinomial, experimentos mostram que QPA, quando comparado aos algoritmos anteriores, reduz consideravelmente a quantidade dos cálculos realizados nos testes de escalonabilidade para quase todas as situações avaliadas [Zhang e Burns 2009].

2.2.2 O Surgimento e a Evolução dos Servidores

Historicamente os servidores foram idealizados para viabilizar a convivência de tarefas periódicas críticas com tarefas aperiódicas não críticas, sem que estas últimas ficassem restritas a executar em *background*, ou seja, nos instantes de ociosidade do sistema. Tarefas que compartilham um mesmo processador podem exercer interferências não desejadas umas nas outras, e a necessidade de isolar tarefas críticas de tarefas não críticas motivou a criação destes mecanismos de reserva de tempo. O propósito é limitar o uso de um processador por um conjunto de tarefas, ou mesmo de garantir uma fração do tempo de um processador para algumas tarefas, promovendo um isolamento temporal.

Segundo Sprunt et al.(1989), antes do surgimento dos mecanismos de reserva de tempo, a estratégia comumente utilizada para viabilizar a convivência entre tarefas periódicas críticas e tarefas aperiódicas não críticas em um mesmo sistema era o escalonamento prioritário das tarefas críticas e a execução das tarefas não críticas em *background*, aproveitando o tempo em que o processador estaria ocioso. O primeiro mecanismo de escalonamento de que se tem conhecimento e que poderia ser propriamente chamado de servidor foi uma extensão do algoritmo *Rate Monotonic* denominado *Polling Server* [Strosnider et al. 1995]. Este servidor foi implementado através da criação de uma tarefa periódica fictícia que era liberada em intervalos regulares de tempo e que, ao invés de executar seu próprio código em um processador quando fosse escalonada, executava em seu lugar qualquer tarefa aperiódica que estivesse pendente. Caso não existissem tarefas aperiódicas pendentes no instante em que o *Polling Server* fosse escalonado, este suspendia a sua execução voltando a ser liberado somente no início do seu próximo período. Neste caso, o tempo não utilizado pelo servidor seria aproveitado pelas tarefas periódicas. As tarefas aperiódicas pendentes não atendidas pelo servidor ainda poderiam

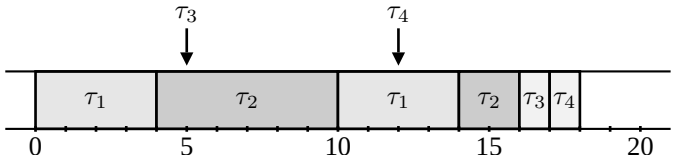


Figura 2.1. Escalonamento *Rate Monotonic* das tarefas periódicas mais prioritárias $\tau_1 : (4, 10)$ e $\tau_2 : (8, 20)$, junto com duas tarefas aperiódicas τ_3 e τ_4 de custo unitário que são liberadas respectivamente nos instantes $t = 5$ e $t = 12$, executando em *background*. [Sprunt et al. 1989]

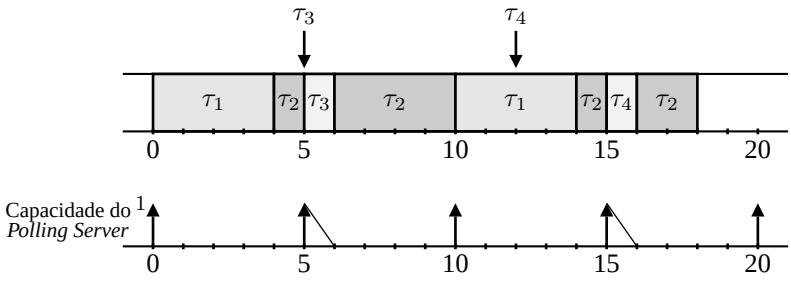


Figura 2.2. Escalonamento *Rate Monotonic* das tarefas periódicas τ_1:$(4, 10)$ e τ_2:$(8, 20)$, junto com um Polling Server que tem tempo de execução unitário, período 5 e atende a duas tarefas aperiódicas τ_3 e τ_4 com custo unitário e liberação nos instantes $t = 5$ e $t = 12$. [Sprunt et al. 1989]

executar em *backgroung*, aproveitando o tempo ocioso do processador.

O comportamento de servidores como o *Polling Server* pode ser definido através de certas regras de comportamento. Em geral, servidores possuem regras que estabelecem a forma como a sua carga é consumida, a frequência com que a sua capacidade de execução é recarregada, e a forma com que o seu *deadline* é calculado.

As Figuras 2.1 e 2.2 ilustram exemplos que apresentam tarefas aperiódicas sendo escalonadas em *background* e com o auxílio de um *Polling Server*, respectivamente. Nestes exemplos, duas tarefas periódicas críticas $\tau_1 : (4, 10)$ e $\tau_2 : (8, 20)$, liberadas em $t = 0$, são escalonadas conjuntamente com duas tarefas aperiódicas τ_3 e τ_4 de custo computacional unitário, liberadas respectivamente nos instantes $t = 5$ e $t = 12$.

Como uma evolução do *Polling Server*, o *Deferrable Server* [Strosnider et al. 1995] igualmente estende o algoritmo *Rate Monotonic*, concebendo um servidor como sendo uma tarefa fictícia que serve tarefas aperiódicas não críticas. A diferença entre os dois servidores é que o *Deferrable Server* preserva a sua capacidade de execução no caso de não haver nenhuma tarefa aperiódica pendente para servir no instante em que ele for escalonado. Ao adiar a sua execução mantendo a sua capacidade, o *Deferrable Server* possibilita que qualquer tarefa aperiódica se utilize desta capacidade assim que for liberada. A cada início de um novo período, a capacidade de execução do *Deferrable Server* é reestabelecida em sua totalidade, podendo este voltar a servir tarefas aperiódicas.

Note que, como na base desta estratégia encontra-se o algoritmo *Rate Monotonic*, quando

o servidor for criado com período inferior ao período de qualquer tarefa periódica, ele será executado em mais alta prioridade repassando esta prioridade para as tarefas aperiódicas a que servir, sem colocar em risco as tarefas periódicas críticas.

Outros servidores, como o *Sporadic Server* [Sprunt et al. 1989] e o *Slack Stealer* [Lehoczky e Ramos-Thuel 1992], mantiveram a estratégia de ter por base o algoritmo *Rate Monotonic*, ampliando a capacidade de processamento oferecido às tarefas aperiódicas. O principal motivo para a não utilização de EDF por estes servidores no atendimento de tarefas aperiódicas não críticas é que EDF utiliza os *deadlines* das tarefas para atribuir a sua prioridade. O TBS (*Total Bandwidth Server*) [Spuri e Buttazzo 1994], diferente dos servidores anteriormente citados, utiliza o algoritmo EDF como base de escalonamento.

Sempre que houver tarefas a serem atendidas, um servidor TBS executa à sua máxima taxa de execução proporcionando o melhor tempo de resposta à tarefa cliente sem o comprometimento da escalonabilidade do conjunto das tarefas periódicas. O limite máximo para a taxa de execução do servidor TBS é estabelecido como sendo $R_s = 1 - R(\Gamma_p)$, onde Γ_p é o conjunto composto pelas tarefas periódicas do sistema. Como toda a capacidade do servidor é entregue ao seu cliente atual, ele somente poderá ser recarregado para o atendimento de uma nova tarefa aperiódica no seu *deadline* corrente ou, no caso de neste instante não haver tarefa aperiódica pendente para execução, na liberação da próxima tarefa aperiódica.

Com a ideia de promover extensões para EDF de servidores originalmente concebidos para funcionar com sistemas de tarefas escalonados por *Rate Monotonic*, o *Deadline Sporadic Server*(DSS) [Ghazalie e Baker 1995] é uma extensão para EDF do *Sporadic Server*. O servidor DSS mantém a principal característica do *Sporadic Server*, que consiste em estabelecer o seu próximo instante de recarga a partir do primeiro uso da sua capacidade de execução. No entanto, para aumentar a prioridade com que seus clientes são executados sem que a sua taxa de execução ultrapasse a um limite estipulado, esta capacidade é fracionada sendo estabelecidos *deadlines* intermediários para cada fração estipulada.

Inspirado pelo comportamento do servidor DSS, que garante executar a uma taxa não superior a um limite máximo, foi concebido o servidor CBS (*Constant Bandwidth Server*) [Abeni e Buttazzo 1998] que limita a sua contribuição à taxa de execução total do sistema mesmo sob uma sobrecarga de tarefas aperiódicas clientes. Um servidor CBS, após sofrer recarga, pode executar seus clientes normalmente até que a sua capacidade de execução se esgote ou até que não haja mais nenhuma tarefa aperiódica pendente para execução. No primeiro caso, o servidor terá a sua capacidade restabelecida na sua totalidade e o seu novo *deadline* passará a ser igual ao seu *deadline* atual D somado à duração de seu período T. No segundo caso, o servidor deverá aguardar o próximo instante r em que uma tarefa aperiódica cliente for liberada. Caso a utilização do restante da sua capacidade de execução no intervalo $[r, D)$ resultar na sua execução a uma taxa não superior ao seu limite máximo R_s, o servidor poderá voltar a atender aos seus clientes normalmente utilizando o restante da sua capacidade de execução e o seu *deadline* atual. Caso contrário, o servidor terá a sua capacidade restabelecida na sua totalidade e o seu

novo *deadline* passará a ser igual a $r + T$.

Os algoritmos apresentados nesta seção resumem os resultados que formam a base do conhecimento construído sobre o problema que envolve o escalonamento em ambientes com um único processador. No entanto, quando o ambiente em que as tarefas devem ser executadas possui dois ou mais processadores, o sistema pode assumir comportamentos inesperados [Andersson e Jonsson 2002]. Com isso, algoritmos ótimos concebidos para o escalonamento em um único processador perdem a sua otimalidade e passam a produzir resultados pouco satisfatórios [Dhall e Liu 1978].

2.3 ANOMALIAS PRESENTES NO ESCALONAMENTO EM MÚLTIPLOS PROCESSADORES

Em sistemas com múltiplos processadores, alterações que supostamente deveriam facilitar o atendimento das restrições temporais do sistema podem provocar o não atendimento de algum *deadline* [Andersson e Jonsson 2000]. Estas *anomalias* dificultam a identificação de um eventual pior caso existente e, conseqüentemente, a verificação da escalonabilidade de um conjunto de tarefas sob um dado algoritmo.

As anomalias podem ser entendidas como o não atendimento de intuições normalmente assumidas no escalonamento de tarefas. Como exemplo, se todos os jobs de uma tarefa atendem aos seus respectivos *deadlines* em um cenário em que as demais tarefas liberam jobs a cada intervalo mínimo entre duas liberações sucessivas, espera-se que estes jobs também atendam aos seus *deadlines* quando os jobs das demais tarefas forem liberados de forma mais espaçada no tempo. Entretanto, não é isso que ocorre.

O efeito Dhall [Dhall e Liu 1978] é um exemplo da ineficácia de EDF e de *Rate Monotonic* globalmente aplicados em sistemas com múltiplos processadores e é ilustrado pela Figura 2.3, onde m tarefas periódicas com períodos curtos ocupam todos os m processadores do sistema impedindo que a tarefa τ_0, de alta taxa de utilização mas com período longo, atenda ao seu *deadline*. Ao escalonar um sistema de tarefas com estas características, com ϵ tendendo a 0, a soma dos tempos de processamento utilizados nos m processadores não seria maior do que o tempo que um único processador poderia ter oferecido. No entanto, bastaria dedicar um processador à execução de τ_0 que todas as demais tarefas poderiam ser escalonadas em um único processador e atender aos seus *deadlines*. O escalonamento apresentado na figura teria sido o mesmo no caso da aplicação do algoritmo *Rate Monotonic*.

Na Figura 2.4 pode-se ver um exemplo de anomalia durante o uso de EDF em um sistema com dois processadores. Observe que, ao selecionar as duas tarefas com o menor *deadline* para executar, o aumento do período de uma tarefa causa a perda do *deadline* de uma outra tarefa do sistema. Considere o sistema com tarefas periódicas e deadline implícito cujos custos de execução e períodos são os seguintes: $\tau_1 : (2, 3)$, $\tau_2 : (2, 4)$, $\tau_3 : (8, 12)$. Quando o período da tarefa τ_1 é ampliado para 4, as tarefas τ_1 e τ_2 passaram a ocupar simultaneamente os dois

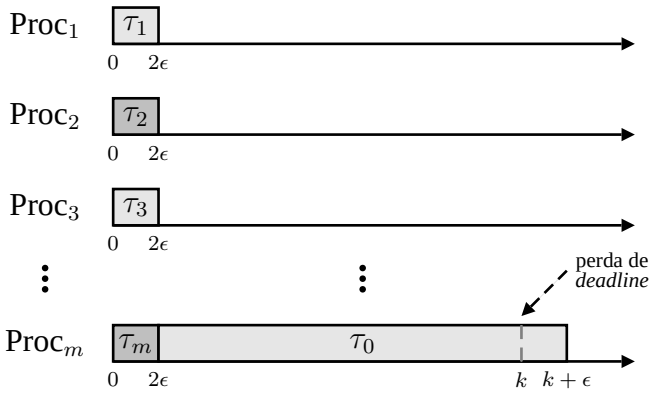


Figura 2.3. Efeito Dhall presente no escalonamento EDF das tarefas $\tau_0 : (k - \epsilon, k)$,$\tau_1 = \tau_2 = \ldots = \tau_m$:$(2\epsilon, l)$ (com $l < k$) em m processadores. Quando $\epsilon \longrightarrow 0$, a utilização dos processadores é limitada em $1/m$.

processadores do sistema, interferindo em τ_3 e fazendo com que esta perca o seu *deadline* no instante $t = 12$.

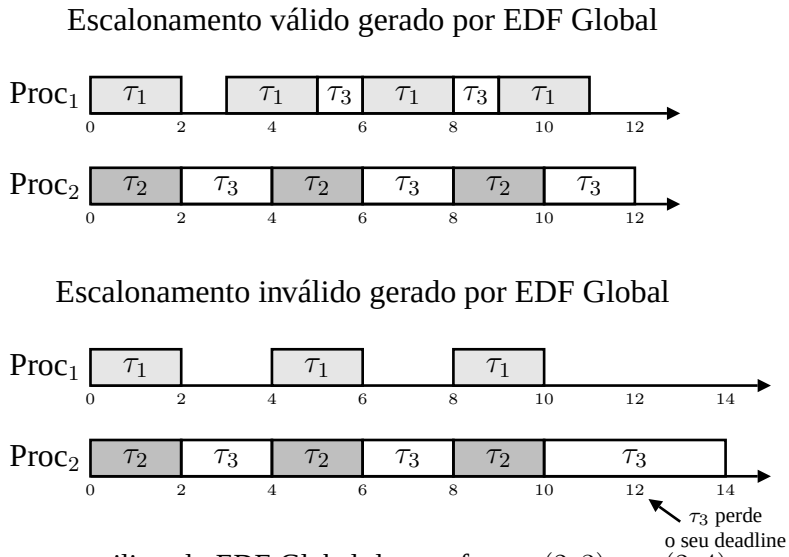


Figura 2.4. Escalonamento utilizando EDF Global das tarefas $\tau_1 : (2, 3)$, $\tau_2 : (2, 4)$, $\tau_3 : (8, 12)$. Postergando o período de τ_1 para 4 provoca perda do *deadline* de τ_3 em 12

Outro exemplo de anomalia, ainda com a utilização de EDF Global, pode ser visto na Figura 2.5. Neste caso, quando o período de uma tarefa é ampliado, ocorre a perda do seu próprio *deadline*. Considere o seguinte conjunto de tarefas: $\tau_1 : (2, 4)$, $\tau_2 : (3, 5)$, $\tau_3 : (7, 10)$. Passando o período da tarefa τ_3 para 11, a liberação do segundo job de τ_3 só ocorrerá uma unidade de tempo mais tarde, não permitindo que o seu próprio *deadline* seja atendido em 22.

Outros exemplos de anomalias foram relatados por Ha e W.S.Liu(1993) e principalmente por Andersson e Jonsson(2002).

Não se deve esperar que os sistemas multiprocessados tenham um comportamento seme-

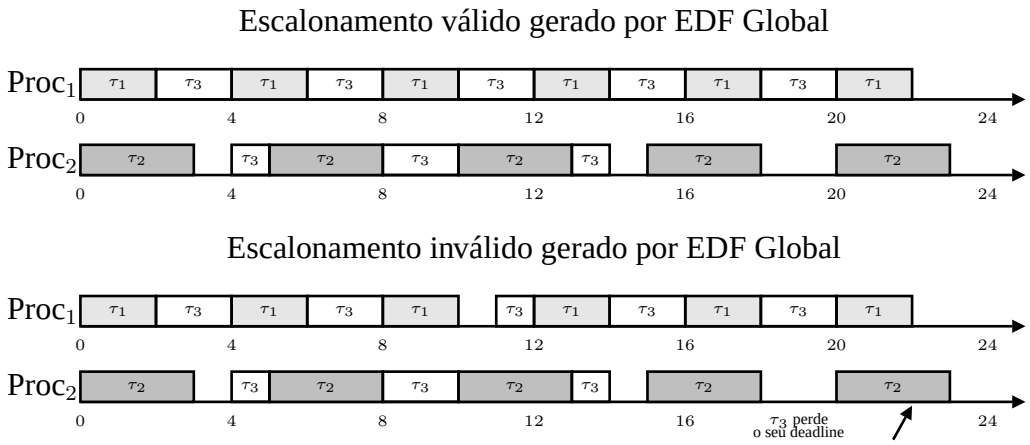


Figura 2.5. Escalonamento utilizando EDF Global das tarefas $\tau_1 : (2, 4)$, $\tau_2 : (3, 5)$, $\tau_3 : (7, 10)$. Postergando o período de τ_3 para 11 provoca a perda do *deadline* do seu próximo job em 22.

lhante ao comportamento dos sistemas com um único processador quando escalonados por algoritmos concebidos para estes últimos. Os vários algoritmos concebidos especificamente para o escalonamento de tarefas em sistemas de tempo real multiprocessados, que serão apresentados nas próximas seções, tentam contornar estas anomalias existentes.

2.4 MECANISMOS DE ESCALONAMENTO PARA MULTIPROCESSADORES

O problema de escalonar um conjunto de tarefas de tempo real em um multiprocessador tem sido extensivamente estudado, e diversas abordagens para este problema vêm sendo propostas. Inicialmente os trabalhos nesta área eram classificados em dois grupos, os algoritmos globais e os particionados [Dhall e Liu 1978]. Mais recentemente, algumas estratégias foram concebidas apresentando características presentes tanto na classe dos algoritmos globais como na classe dos algoritmos particionados. Essas estratégias ficam em uma classe intermediária, compondo o conjunto dos algoritmos semi-particionados.

Os algoritmos da classe das abordagens particionadas associam as tarefas estaticamente aos processadores sem permitir que estas migrem de um processador para outro, sendo a decisão de quais tarefas devem ser executadas em quais processadores tomada por um algoritmo de *bin-packing*. Apesar de normalmente os custos com as migrações serem considerados nulos para a análise dos algoritmos de escalonamento, a realidade é que estes custos existem e quanto mais migrações forem necessárias, maior será o tempo de processamento perdido no gerenciamento dessas migrações. Portanto é desejável que um algoritmo de escalonamento provoque o menor número de migrações possível. Assim, o particionamento estático das tarefas do sistema em subgrupos, cada qual associado com um processador, impede que haja migrações e permite que os algoritmos de escalonamento com otimalidade comprovada quando aplicados no escalonamento em um único processador possam ser aplicados. Infelizmente, é pouco provável que abordagens particionadas consigam lidar com um conjunto de tarefas que utilizem todo o tempo

disponível de todos os processadores de um sistema [Koren et al. 1998] e, portanto, soluções particionadas não são uma opção quando a otimalidade for exigida para sistemas com mais de um processador.

Apesar da abordagem particionada ser interessante por poder aplicar os algoritmos ótimos de escalonamento em cada processador individualmente, a associação fixa de tarefas com processadores é um problema NP-Difícil [Garey e Johnson 1979]. Além deste problema, já foi comprovado que alguns sistemas de tarefas não são escalonáveis se a migração de tarefas não for permitida [Carpenter et al. 2004]. Como exemplo, se for considerado um sistema com m processadores e $m + 1$ tarefas com utilização igual a $0.5 + \epsilon$, onde m é um valor muito grande e ϵ é um valor muito pequeno, pode-se associar apenas uma única tarefa a cada processador e ainda sobrará uma tarefa. Em outras palavras, não será possível escalonar este sistema mesmo com uma utilização total próxima de apenas 50%.

A classe das abordagens globais é composta pelas estratégias que permitem a migração de tarefas entre processadores. Normalmente estes esquemas são baseados em uma fila única e global de tarefas, com a atribuição das tarefas pelos processadores disponíveis sendo realizada em tempo de execução. Em alguns algoritmos que permitem a migração de tarefas, cada job de uma tarefa deve ser executado inteiramente em um mesmo processador. Entretanto, a maioria dos algoritmos globais permite que a execução de um job seja suspensa em um processador e posteriormente retomada em um outro processador. Políticas de escalonamento baseadas em prioridades [Baker 2003] [Baker et al. 2008] [Baruah e Baker 2008] [Piao et al. 2006] podem ser utilizadas na ordenação desta fila, embora habitualmente apresentem um limite baixo de escalonabilidade.

Trabalhos recentes propuseram uma abordagem diferente do problema que busca unir as virtudes das abordagens globais e particionadas. Nessa nova abordagem, denominada semi-particionada, as tarefas são alocadas estaticamente nos processadores, sendo permitido que apenas algumas tarefas possam migrar, de forma controlada, entre processadores. A quantidade de tempo e os processadores que executarão as tarefas migratórias e não migratórias são definidos em tempo de projeto. Esta estratégia foi adotada pela primeira vez pelo algoritmo EDF-fm [H.Anderson et al. 2005], que tinha como objetivo escalonar tarefas não críticas de tempo real. Apesar de alguns dos algoritmos que se seguiram ao EDF-fm terem sido concebidos utilizando esquemas com prioridade fixa, a maioria dos algoritmos semi-particionados elaborados para sistemas críticos de tempo real utiliza o algoritmo EDF como base. Para tarefas de *deadline* implícito, os principais resultados da abordagem semi-particionada foram obtidos pelos algoritmos EKG periódico [Andersson e Tovar 2006] e esporádico [Andersson e Bletsas 2008], além das abordagens *Notional Processors* [Bletsas e Andersson 2009] e Hime [Santos-Jr. et al. 2013]. Já as tarefas esporádicas com deadline arbitrário foram tratadas pelos algoritmos EDF-SS [Andersson et al. 2008] e EDF-WM [Kato et al. 2009], através do uso de uma análise pseudo-polinomial da demanda do sistema de tarefas, e pelo algoritmo C=D [Burns et al. 2012]. Davis e Burns(2011) oferecem uma visão geral dos principais trabalhos sobre escalonamento

em sistemas multiprocessados.

2.4.1 Abordagens Globais

Fisher(2007) provou não ser possível a concepção de um algoritmo que escalone qualquer sistema factível composto por tarefas esporádicas com *deadline* restrito ou arbitrário. No entanto, para tarefas periódicas com *deadline* implícito, alguns algoritmos globais conseguem atingir esta otimalidade. Entre eles, encontram-se o algoritmo *Proportionate Fairness* (*Pfair*) [Baruah et al. 1996], algumas de suas variantes como PD [Baruah et al. 1995] e ER-PD [Anderson e Srinivasan 2000], e o algoritmo *Largest Local Remaining Execution Time First* (LLREF) [Cho et al. 2006]. Apesar da otimalidade teórica destas abordagens, o uso prático destes algoritmos em sistemas reais é potencialmente problemático devido ao alto custo computacional ocasionado pela excessiva quantidade de migrações e preempções geradas [Davis e Burns 2011]. Para o caso particular de sistemas periódicos em que todas as tarefas compartilham os mesmos *deadlines* absolutos, o algoritmo de McNaughton [McNaughton 1959] produz um escalonamento ótimo de forma simples e eficiente, apesar de somente poder ser utilizado com este modelo de tarefas bastante restrito.

2.4.1.1 Algoritmo de McNaughton

O Algoritmo de McNaughton [McNaughton 1959] é um algoritmo ótimo no sentido de que é capaz de escalonar qualquer sistema com tarefas periódicas com utilização não superior a m e que compartilhem os mesmos *deadlines*, causando um máximo de $m - 1$ preempções entre cada dois *deadlines* consecutivos.

Como todos os jobs compartilham o mesmo *deadline* D, o algoritmo de McNaughton lista os jobs em uma ordem arbitrária, somando os seus custos computacionais. Esta listagem é interrompida quando a soma atinge o valor de D, definindo a sequência de jobs que será executada no primeiro processador. A listagem é reiniciada do ponto em que foi interrompida, parando novamente quando a soma dos custos computacionais atingir $2D$. Esta será a sequência de jobs a ser executada no segundo processador e o mesmo procedimento deve ser repetido até que a sequência de jobs de todos os processadores seja definida. Note que, quando a execução de um job não couber inteiramente em um processador i, o seu complemento será executado no processador $i + 1$, como pode ser visto na Figura 2.6.

Embora o algoritmo de McNaughton seja ótimo no escalonamento de tarefas com mesmo *deadline*, a sua utilização prática é bastante limitada por poder ser aplicado somente em um modelo muito restritivo de tarefas .

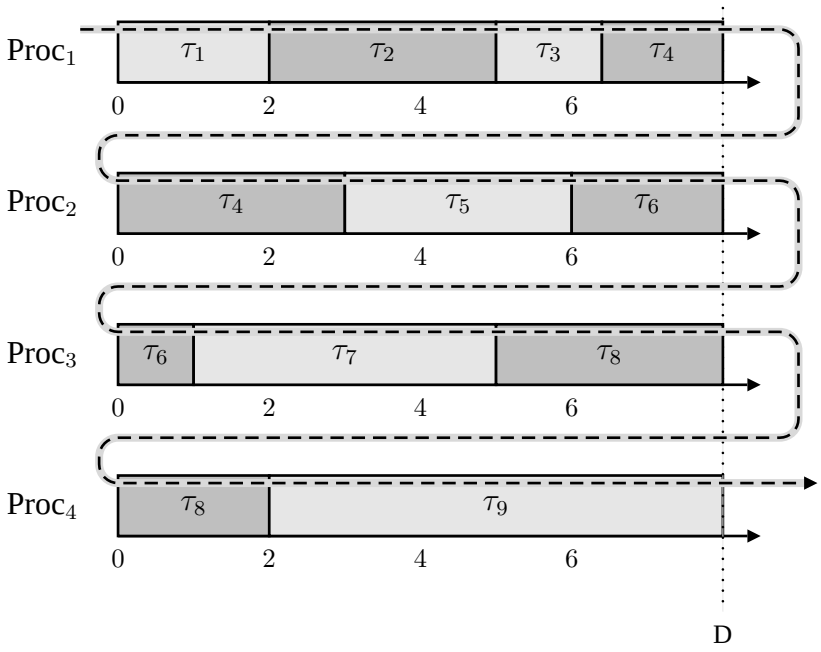


Figura 2.6. Algoritmo de McNaughton escalonando nove tarefas que compartilham o mesmo *deadline* D

2.4.1.2 *Proportionate Fairness*

O *Pfair* (*Proportionate Fairness*) [Baruah et al. 1996] introduz a ideia de um escalonamento fluido em que as tarefas executam ao longo do tempo de forma proporcional a sua taxa de execução. Antes de ser propriamente um algoritmo de escalonamento, o *Proportionate Fairness* é uma propriedade concebida por Baruah et al.(1996) que garante a otimalidade no escalonamento de tarefas periódicas com *deadline* implícito. A Figura 2.7 apresenta o escalonamento de uma tarefa comparada com o seu escalonamento fluido ao longo do tempo.

Em um algoritmo de escalonamento com a propriedade de *Proportionate Fairness*, o tempo é seccionado em *slots* arbitrariamente pequenos e de igual duração. As decisões de escalonamento são tomadas em cada *slot* t, sendo um escalonamento *Sched* aplicado a um conjunto de tarefas Γ em m processadores definido como uma função $\mathit{Sched}: \Gamma \times \mathbb{N} \longmapsto \{0, 1\}$, com $\sum_{\tau_i \in \Gamma} \mathit{Sched}(\tau_i, t) \leq m$, onde uma tarefa τ_i será executada em um processador no instante t se e somente se $\mathit{Sched}(\tau_i, t) = 1$. Para que um escalonamento *Sched* possua a propriedade de *Proportionate Fairness*, ele deve garantir que toda tarefa do sistema não esteja a uma distância maior do que um grânulo de tempo do seu escalonamento fluido ideal, seja a sua frente ou atrás dele.

Baruah et al.(1996) também apresentou um algoritmo, originalmente denominado PF, possuidor da propriedade de *Proportionate Fairness*. Ambos, algoritmo e propriedade, vêm sendo indistintamente referenciados na literatura como *Pfair* e, portanto, adotaremos este mesmo procedimento a partir deste ponto. O algoritmo *Pfair* inicialmente decide escalonar todas as tarefas

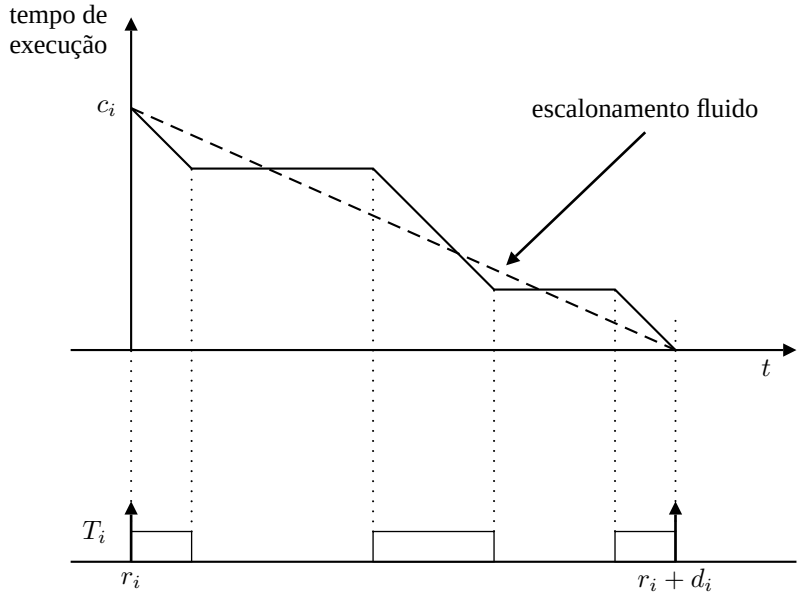


Figura 2.7. Ilustração de um escalonamento fluido [Cho et al. 2006]

que, caso não executem durante o grânulo de tempo atual, passarão a ter uma distância do seu escalonamento fluido superior a um grânulo de tempo. De forma semelhante, o algoritmo decide não escalonar todas as tarefas que, caso executem durante o grânulo de tempo atual, ficarão à frente do seu escalonamento fluido por uma distância superior a um grânulo de tempo. escalonamento das demais tarefas é decidido por um critério secundário de desempate.

Apesar de possuir otimalidade teórica, a potencial adoção de *slots* com tamanho arbitrariamente pequenos e a consequente geração de uma quantidade arbitrariamente grande de preempções e de migrações faz com que a aplicação do *Pfair* em sistemas reais seja inviável, sendo indicado pelos próprios idealizadores que seria interessante a pesquisa de outros algoritmos aplicáveis a sistemas periódicos que minimizassem a quantidade de preempções.

> *Our Pfair scheduling algorithm produces schedules with a large number of preemptions. It would be interesting to investigate algorithms for solving the periodic scheduling problem which minimize the number of preemptions.*
>
> *Baruah et al.(1996)*

2.4.1.3 Variantes do *Pfair*

Após o surgimento do *Pfair*, foram propostas novas estratégias que utilizam o mesmo princípio de escalonamento fluido, mas buscam uma redução na quantidade de preempções. Entre os principais representantes destas estratégias estão os algoritmos PD e ER-PD.

Algoritmo PD

O algoritmo PD (*Pseudo-Deadline Algorithm*) [Baruah et al. 1995] também faz uso da propriedade de *Proportionate Fairness* para resolver o problema do escalonamento de tarefas periódicas em múltiplos processadores. Este algoritmo utiliza inicialmente o mesmo conjunto de critérios utilizados pelo *Pfair* para identificar as tarefas que deverão ser escalonadas, ou que deverão não ser escalonadas, em um dado *slot*. A diferença entre estes dois algoritmos encontra-se nos critérios de desempate para as tarefas cujo escalonamento não puder ser decidido por meio dos critérios iniciais.

Em PD, as tarefas são classificadas como *heavy* (tarefas com utilização superior a $0,5$) ou *light* (tarefas com utilização inferior a $0,5$), podendo uma tarefa com utilização exatamente igual a $0,5$ ser arbitrariamente classificada tanto como *heavy* quanto como *light*. As tarefas *heavy* e *light* são selecionadas para execução seguindo um critério que utiliza a taxa de execução para as tarefas classificadas como *light*, enquanto que para as tarefas classificadas como *heavy* é utilizado o "complemento de um" da sua taxa de execução, ou seja, a taxa proporcional ao tempo em que não deverão utilizar nenhum processador ($1 - \mathrm{R}(\tau)$).

Algoritmo *ER-PD*

Os algoritmos *Pfair* e PD realizam o equivalente a uma divisão das tarefas em partes que devem ser executadas em uma quantidade fixa de *slots*. Uma vez que uma das partes de uma tarefa é concluída, a tarefa é impedida de continuar executando até que a próxima sequencia de slots (onde será executada a próxima parte da tarefa) se inicie. Portanto, os algoritmos *Pfair* e PD se caracterizam por não ser *work conserving*, ou seja, permitem que processadores fiquem ociosos mesmo que ainda haja tarefas a serem executadas. O algoritmo *ER-PD* (*Early-Release Pseudo Deadline*) [Anderson e Srinivasan 2000] é um algoritmo que segue os mesmos princípios do *Pfair* com a característica adicional de ser *work conserving*. Na prática, a única diferença entre os algoritmos PD e ER-PD é que neste último existe a permissão para que a execução de uma tarefa esteja à frente do seu escalonamento fluido por mais do que um grânulo de tempo.

2.4.1.4 *Largest Local Remaining Execution Time First* (LLREF)

O LLREF [Cho et al. 2006], ou *Largest Local Remaining Execution Time First*, é um algoritmo não *work conserving* que aumenta o tamanho dos *slots* de tempo utilizados com o intuito de evitar o excesso de migrações e preempções. Neste algoritmo, são criados planos T-L (*Time and Local Execution Time Domain Plane*) (Figura 2.8), construídos entre cada dois eventos de escalonamento consecutivos, que apresentam graficamente o escalonamento de todas as tarefas do sistema. O fluxo do tempo é representado pelo eixo x e o tempo ainda necessário para a conclusão de uma tarefa é representado no eixo y. Cada tarefa τ_i tem o seu escalonamento fluido até o próximo evento de escalonamento do sistema representado por linhas diagonais. A

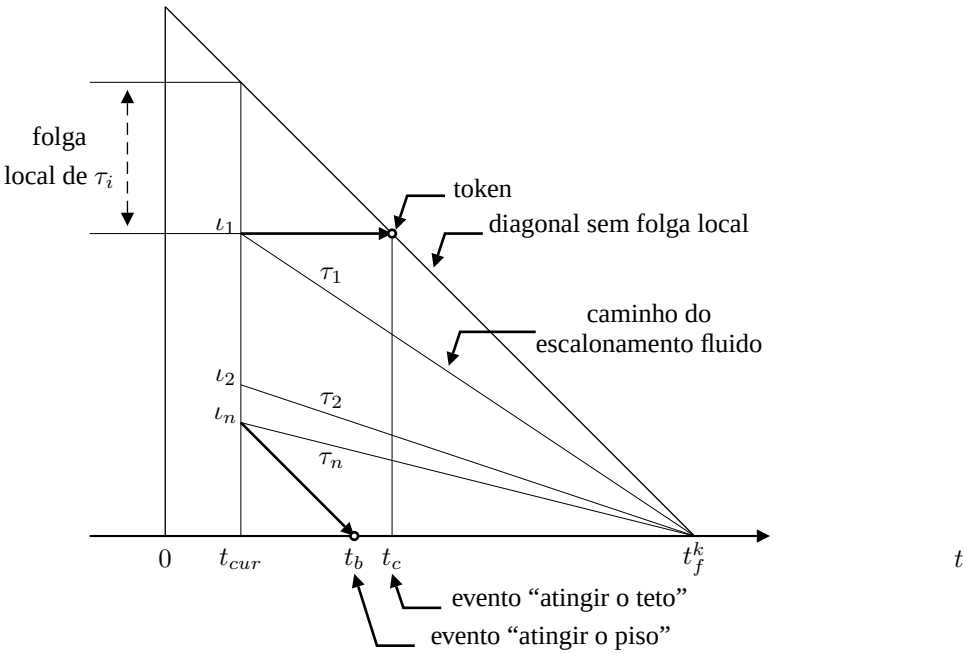


Figura 2.8. k-ésimo Plano TL [Cho et al. 2006]

ideia do LLREF é que, entre cada dois eventos de escalonamento consecutivos, toda tarefa execute a mesma quantidade de tempo que executaria sob um escalonamento fluido.

Em um plano T-L, a situação de uma tarefa é representada por um *token* que descreve a quantidade de tempo que ainda resta para esta tarefa atingir ao seu escalonamento fluido. Pode-se notar na Figura 2.8 que a curva criada pelo *token* decresce a 45 graus durante a execução de sua tarefa, enquanto a não execução da tarefa provoca um deslocamento horizontal no *token*. É possível notar que dois eventos distintos podem ocorrer: um *token* pode chocar-se com a diagonal que representa a não existência de folga local, o que equivale dizer que se esta tarefa não iniciar a sua execução imediatamente ela não alcançará o seu escalonamento fluido no próximo evento de escalonamento do sistema; e um *token* pode chocar-se com o eixo horizontal, o que equivale dizer que esta tarefa já executou tudo o que o seu escalonamento fluido exigiria até o próximo evento de escalonamento do sistema e não deve mais ser executada até lá.

Apesar de conseguir o alongamento no tamanho dos *slots*, o LLREF ainda pode provocar uma alta quantidade de preempções já que os eventos de escalonamento de uma tarefa (liberações e *deadlines* de jobs) interferem em todas as demais tarefas. Desta forma, embora algumas abordagens globais consigam atingir a otimalidade teórica, esta propriedade não vem sendo convertida em resultados práticos para sistemas reais. Como a abordagem particionada possui a limitação impingida pelo problema do empacotamento de tarefas em processadores, recentemente surgiram novas ideias voltadas a uma abordagem intermediária, denominada semi-particionada.

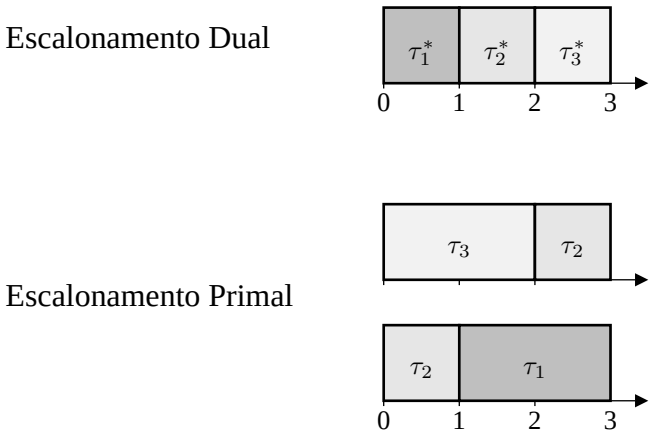


Figura 2.9. Escalonamento Primal x Dual das tarefas τ_1:$(2,3)$, τ_2:$(2,3)$ e τ_3:$(2,3)$

2.4.1.5 *Reduction to Uniprocessor* (RUN)

O algoritmo RUN (*Reduction to Uniprocessor*) [Regnier et al. 2011, Regnier et al. 2012] utiliza uma abordagem diferente para resolver o problema do escalonamento de sistemas com tarefas periódicas em multiprocessadores. Ele reduz este problema ao escalonamento de um sistema equivalente em um único processador e, posteriormente, produz uma solução com o uso de EDF, atingindo a otimalidade para esta classe de problemas.

Para um sistema com n tarefas, decidir quais as tarefas que devem ocupar um conjunto composto por m processadores durante certo período é equivalente a decidir quais as $n - m$ tarefas que não devem executar durante este mesmo período. A Figura 2.9 ilustra este conceito, exibindo o escalonamento das tarefas $\tau_1:(2,3)$, $\tau_2:(2,3)$ e $\tau_3:(2,3)$ em $m = 2$ processadores. Como τ_1, τ_2 e τ_3 devem executar por duas unidades de tempo no intervalo $[0,3)$, isto significa que cada uma delas não deverá executar por uma unidade de tempo neste mesmo período, o que pode ser representado pelas respectivas tarefas duais $\tau_1^*:(1,3)$, $\tau_2^*:(1,3)$ e $\tau_3^*:(1,3)$. O escalonamento destas tarefas duais em um processador virtual indica os intervalos em que cada uma das tarefas não deverá executar, induzindo o escalonamento real, como apresentado na figura.

Desta forma, RUN reduz um sistema composto por m processadores em um sistema com $n - m$ processadores virtuais. Esta redução só é interessante quando $n - m < m$ pois, neste caso, o sistema equivalente possuirá menos processadores do que o sistema original, resultando em menos decisões de escalonamento a cada instante. Assim, antes de proceder a redução, RUN agrupa as tarefas com baixa utilização associando cada grupo com um servidor que terá uma utilização mais alta, garantindo um sistema dual com menor quantidade de processadores. Este procedimento é repetido recursivamente até que um sistema uniprocessado seja obtido e resolvido por EDF.

2.4.1.6 Algoritmo U-EDF

O U-EDF (*Unfair scheduling algorithm based on EDF*) [Nelissen et al. 2012] é um algoritmo de escalonamento para sistemas de tarefas esporádicas com *deadline* implícito que foi inicialmente concebido para tarefas periódicas [Nelissen et al. 2011]. Ele tem por característica não ter por base a propriedade de *fairness*. A ideia central é alocar os jobs a um processador conforme estes forem sendo liberados, ocupando o processador de forma proporcional à sua utilização. Quando uma tarefa liberar um job e este não puder mais ser atendido totalmente pelo processador atual, o job é dividido proporcionalmente entre este processador e o processador seguinte, iniciando a ocupação deste último. Os jobs liberados pelas próximas tarefas serão alocados neste novo processador, de forma semelhante ao que foi feito com o processador anterior. O escalonamento dos jobs ocorre por uma variação do EDF que garante que duas partes de um mesmo job não executem simultaneamente em processadores distintos.

O U-EDF foi o primeiro algoritmo ótimo não baseado em *fairness* para o escalonamento de tarefas esporádicas com *deadline* implícito em múltiplos processadores, no sentido em que ele é capaz de escalonar qualquer sistema que tenha utilização total não superior a m em um ambiente com m processadores.

2.4.2 Abordagens Semi-Particionadas

A primeira referência de que se tem conhecimento sobre estratégias que buscam uma abordagem intermediária entre a abordagem particionada e a abordagem global foi o algoritmo EDF-fm [H.Anderson et al. 2005]. Entretanto o objetivo deste algoritmo não era escalonar tarefas de tempo real críticas, mas sim tarefas não críticas. O primeiro algoritmo semi-particionado com bons resultados para tarefas críticas de que se tem conhecimento foi o algoritmo EKG, que tinha por objetivo escalonar sistemas de tarefas com *deadline* implícito que fossem periódicas [Andersson e Tovar 2006] e esporádicas [Andersson e Bletsas 2008].

2.4.2.1 Algoritmo EKG

O algoritmo EKG (Edf with task splitting and K processors in a Group) [Andersson e Tovar 2006] foi concebido inicialmente para escalonar sistemas de tarefas periódicas com *deadline* implícito. As tarefas do sistema são classificadas em tarefas pesadas e em tarefas leves de acordo com a sua respectiva utilização, sendo que cada tarefa pesada receberá um processador dedicado à sua execução. As tarefas leves são alocadas sequencialmente aos processadores não dedicados, sendo utilizado o teste de escalonabilidade para EDF (utilização total não superior a 1) para decidir se uma tarefa ainda pode ser atendida inteiramente pelo processador atual ou não. Caso uma tarefa falhe em ser alocada inteiramente em um processador e este seja o último de um grupo de K processadores, a tarefa será alocada inteiramente no processador seguinte. No caso do teste falhar para uma tarefa em um processador que não seja o último de um grupo, esta

tarefa será parcialmente alocada ao processador atual e parcialmente alocada ao processador seguinte, migrando durante a sua execução.

Tarefas migratórias são divididas em partes que executam proporcionalmente em intervalos definidos por duas liberações sucessivas no sistema. Cada uma destas partes é ainda subdividida em duas outras partes que executarão em processadores distintos com prioridade máxima. Para que a execução destas partes não se sobreponham no tempo, uma delas será executada em uma reserva de tempo localizada no início e a outra será executada em uma reserva de tempo localizada no final do respectivo intervalo. As demais tarefas são escalonadas por EDF.

Através do parâmetro K, pode-se obter o limite máximo para a utilização total dos processadores do sistema, denominado SEP, calculado através da Equação 2.7 e utilizado para a classificação das tarefas leves e pesadas. O limite máximo para a média de preempções provocadas pelo EKG é de $2K$ por job.

$$SEP = \begin{cases} \frac{K}{K+1} & \text{Se } K < m \\ 1 & \text{Se } K = m \end{cases} \tag{2.7}$$

Com a escolha de $K = 2$, algoritmo EKG consegue atingir a uma utilização total de $SEP = \frac{K}{K+1} = 66\%$ dos processadores, garantindo que a quantidade média de preempções por job processado não seja superior a $2K = 4$. Com $K = m$, a utilização total dos processadores pelo EKG chega a 100%, no entanto ele pode provocar até $2m$ preempções por job.

Uma variação do EKG foi apresentada por Andersson e Bletsas(2008) para sistemas com tarefas esporádicas. Nesta variante, o tempo é dividido em *slots* de tamanhos fixos com duração $S = TMIN/\delta$, onde $TMIN$ é o menor período entre todas as tarefas do sistema, δ é um parâmetro de entrada para o algoritmo, e SEP passa a ser calculado em função do δ escolhido. Quanto maior for o valor escolhido para δ, maior será o limite da utilização dos processadores do sistema, porém também maior será o número de preempções provocadas. Para o caso em que se admite o menor número de preempções ($\delta = 1$), esta variante é capaz de escalonar sistemas com até 65.7% de utilização. Nesta variante do EKG, as reservas de tempo utilizadas para a execução das tarefas migratórias são infladas para que a escalonabilidade do sistema seja garantida.

2.4.2.2 Algoritmo *Notional Processors*

O algoritmo *Notional Processors* [Bletsas e Andersson 2009], idealizado para sistemas de tarefas esporádicas com *deadline* implícito, cria o conceito de um processador lógico que é, na verdade, um mapeamento sobre um conjunto de períodos ociosos em processadores físicos. Portanto, se uma tarefa for escalonada em um *Notional Processors* π' em um instante t, ela estará efetivamente sendo executada em algum processador físico π cujo mapeamento associa π' e π no instante t.

Este algoritmo pode ser entendido como uma sequencia de três etapas. Inicialmente, as tarefas são alocadas aos processadores de forma que as tarefas que possuam alta utilização sejam alocadas antes das tarefas com baixa utilização. Isto é feito até que não existam tarefas que ainda possam ser alocadas em algum processador. Após a etapa de alocação das tarefas, o tamanho do *slot* de tempo S utilizado em todo o sistema é definido como sendo o menor valor entre os períodos de todas as tarefas já alocadas dividido por um fator arbitrário δ. O correto ajuste no valor de δ permite um balanceamento entre o tempo de processamento possível de ser utilizado e a quantidade de preempções provocadas. Para cada processador físico, é calculada uma reserva de tempo (fração de tempo do *slot*) que seja suficiente para que as tarefas alocadas naquele processador atendam aos seus *deadlines* se forem escalonadas por EDF. O cálculo feito para esta reserva é o mesmo utilizado na variante do EKG para tarefas esporádicas [Andersson e Bletsas 2008], apresentada ao final da Seção 2.4.2.1. As reservas criadas são organizadas dentro de cada *slot* de forma que o início do período ocioso em um processador coincida com o término do período ocioso em outro processador, compondo os *Notional Processors* que serão utilizados pelas tarefas ainda não alocadas em processadores. É interessante observar que, enquanto no algoritmo EKG as reservas delimitam a execução das tarefas migratórias, em *Notional Processors* as reservas delimitam a execução das tarefas que não migram.

A análise apresentada por Bletsas e Andersson(2009) mostra que o algoritmo *Notional Processors* é capaz de escalonar sistemas com até 66.6% provocando menos preempções do que a versão do algoritmo EKG para tarefas esporádicas consegue escalonar sistemas com até 65.7%.

2.4.2.3 Algoritmo EDF-SS

Seguindo um princípio semelhante ao EKG e ao Notional Processors, mas voltado para tarefas esporádicas com *deadline* arbitrário, o algoritmo EDF-SS (*EDF scheduling of non-split tasks with Split tasks scheduled in Slots of duration* DTMIN$/\delta$) [Andersson et al. 2008] divide o tempo em *slots* com duração $S = \text{DTMIN}/\delta$, onde DTMIN é o menor valor entre os períodos e os *deadlines* das tarefas do sistema e δ é um parâmetro arbitrariamente escolhido. As tarefas são distribuídas pelos processadores e as tarefas que não conseguirem ser alocadas inteiramente em nenhum processador são divididas em duas partes que executarão em duas reservas de tempo localizadas no início e no final de cada *slot*, garantindo que estas partes não executarão simultaneamente.

Para distribuir as tarefas pelos processadores, organiza-se as tarefas em ordem decrescente de *deadline* e aloca-se tantas tarefas quanto possível em um processador atual. Um teste particular que executa em tempo pseudo-polinomial, derivado a partir da função Dbf [Baruah et al. 1990b], é utilizado para saber se uma tarefa pode ser adicionada a um processador sem ameaçar a escalonabilidade das tarefas já alocadas a este mesmo processador. Note que, no instante em que este teste de escalonabilidade é aplicado, já foi feita a reserva utilizada para a execução da tarefa que migra entre o processador anterior e o processador atual. Uma vez que nenhuma

outra tarefa possa ser inteiramente alocada ao processador atual, a tarefa com o menor *deadline* é selecionada e dividida entre o processador atual e o próximo processador. As durações das reservas estabelecidas para cada tarefa são calculadas utilizando-se a mesma função derivada da função Dbf.

2.4.2.4 Algoritmo EDF-WM

O algoritmo *EDF with Window constraint Migration* [Kato et al. 2009], ou EDF-WM, também utiliza uma abordagem semi-particionada para escalonar tarefas esporádicas com *deadline* arbitrário em multiprocessadores. Nele, uma tarefa migratória é dividida em duas ou mais partes, sendo que a execução em paralelo de duas partes quaisquer da mesma tarefa é impedida pela definição de janelas de tempo em que cada parte poderá executar.

O EDF-WM inicialmente organiza as tarefas em ordem decrescente de utilização e as distribui pelos processadores através da heurística *first-fit*. Não sendo mais possível alocar tarefas inteiras aos processadores, as tarefas restantes são definidas como migratórias. Para cada tarefa migratória, são feitas sucessivas tentativas de divisão até que se consiga dividir a tarefa na menor quantidade possível de partes. Em cada tentativa, a tarefa é dividida em k partes, sendo que a cada parte será atribuído um *deadline* correspondente a $1/k$ parte do *deadline* da tarefa original e um instante de liberação igual ao *deadline* da parte imediatamente anterior. O custo computacional atribuído a cada parte é calculado utilizando a função Dbf [Baruah et al. 1990b], que executa em tempo pseudo-polinomial e foi apresentada na Seção 2.2. Se o custo computacional total atribuído às partes da tarefa for inferior ao custo computacional da tarefa, então esta precisará ser dividida em mais partes e uma nova tentativa de dividi-la será feita.

Em experimentos realizados pelos autores com tarefas esporádicas de *deadline* arbitrário, o algoritmo EDF-WM conseguiu escalonar sistemas que utilizavam até 90% dos processadores em aproximadamente 90% dos casos simulados, desempenho que é um pouco inferior ao desempenho do EDF-SS em termos de escalonabilidade. No entanto, os mesmos experimentos mostraram o EDF-WM provocando uma quantidade de preempções e migrações aproximadamente dezesseis vezes menor do que o EDF-SS.

2.4.2.5 Algoritmo C=D

O algoritmo C=D [Burns et al. 2012] trata de tarefas esporádicas com *deadline* arbitrário. A ideia principal é permitir apenas uma tarefa migratória em cada processador, aceitando que tarefas migrem por dois ou mais processadores. Evita-se a execução em paralelo de duas partes de uma mesma tarefa através da fixação do período em que ela executará em cada processador. O *deadline* de cada parte da tarefa é encurtado, passando a ser igual ao seu custo computacional, e a parte seguinte somente é liberada após o *deadline* da primeira parte. Como cada parte das tarefas migratórias devem executar em intervalos específicos de tempo, estas devem ser escalonadas com uma prioridade mais alta do que as demais tarefas do sistema.

Este algoritmo inicia com a distribuição das tarefas pelos processadores, garantindo a escalonabilidade em cada processador pela análise da sua utilização total. Após esta distribuição inicial, o algoritmo C=D escolhe tarefas para migrar, dividindo-as em partes. O tempo de execução da primeira parte da tarefa é fixado com o auxílio do teste de escalonabilidade QPA [Zhang e Burns 2009], um aprimoramento do Dbf, garantindo a escalonabilidade do sistema por EDF e fechando o conjunto de tarefas que executarão neste processador. Este procedimento é repetido até que todas as tarefas sejam distribuídas pelos processadores ou até que não seja possível alocar uma tarefa, caso este em que o algoritmo falha.

Embora não tenha sido apresentada pelos seus idealizadores uma condição de escalonabilidade baseada em utilização, Santos-Jr. et al.(2013a) apresenta uma variante do C=D que garante o escalonamento para conjuntos de tarefas que utilizem não mais do que 72.2% dos processadores de um sistema.

2.4.2.6 Algoritmo Hime

O algoritmo Hime [Santos-Jr. et al. 2013] possui uma abordagem semelhante ao algoritmo C=D para escalonar tarefas esporádicas com *deadline* implícito. As tarefas com permissão para migrar são executadas em mais alta prioridade e podem ser executadas em dois ou mais processadores, sendo que é permitida somente uma tarefa migratória por processador. A principal diferença entre estes dois esquemas consiste no fato de Hime utilizar um teste baseado em utilização para garantir a escalonabilidade em cada processador, ao invés de utilizar testes de escalonamento exatos, como o QPA. Esta abordagem faz com que Hime apresente resultados experimentais um pouco inferiores ao C=D em termos de quantidade de sistemas escalonados, apesar de não existir relação de dominância entre Hime e C=D. Hime garante o escalonamento de qualquer sistema de tarefas com utilização total não superior a 75%.

RESUMO

Algoritmos de escalonamento podem atribuir uma prioridade constante para todos os jobs de uma mesma tarefa, ou permitir que esta prioridade seja alterada com o passar do tempo. Estas duas características dividem estes algoritmos em duas classes distintas: os algoritmos com prioridade fixa e os algoritmos com prioridade dinâmica. Os principais representantes destas classes no escalonamento em ambientes com um único processador são respectivamente os algoritmos *Rate Monotonic* e o algoritmo EDF (*Earliest Deadline First*), sendo este último um algoritmo ótimo no sentido em que se um outro algoritmo de escalonamento produzir um escalonamento válido para um determinado sistema de tarefas, o algoritmo EDF também será capaz de produzir um escalonamento válido para este mesmo sistema de tarefas. Para que um sistema de tarefas periódicas com *deadline* implícito seja escalonável por EDF, basta que a utilização total do sistema não ultrapasse 100%. Para os demais sistemas, o teste exato de demanda QPA pode ser utilizado para garantir a produção de um escalonamento válido por

EDF.

Apesar de terem bons resultados em ambientes com um único processador, os algoritmos concebidos para estes ambientes provocam certas anomalias quando utilizados em ambientes com múltiplos processadores. Assim, surgem os algoritmos específicos para multiprocessadores que podem ser classificados nas abordagens particionada, global e semi-particionada. Os algoritmos particionados podem se utilizar dos resultados consolidados do escalonamento para um único processador, entretanto eles enfrentam como obstáculo um baixo limite para a utilização dos processadores. Alguns algoritmos globais conseguem atingir a otimalidade ao custo de uma grande quantidade de migrações e preempções, o que torna inviável a sua implementação. Algoritmos semi-particionados formam uma abordagem mais recente que busca se utilizar das qualidades presentes nas abordagens anteriores, entretanto elas têm o desafio de garantir que uma mesma tarefa não seja executada em mais de um processador simultaneamente.

Nas diversas estratégias que adotam a abordagem semi-particionada, mecanismos de reserva de tempo tornam-se necessários para evitar que uma mesma tarefa seja escalonada em mais de um processador simultaneamente. Na prática, pode-se entender cada um destes mecanismos de reserva de tempo como um servidor σ que reserva parcelas do tempo de um processador para a execução de um conjunto de tarefas identificadas como suas clientes, denotado por $\mathrm{cli}(\sigma)$. No Capítulo 3 serão apresentados dois novos servidores desenvolvidos neste trabalho que controlam tarefas que executam em mais de um processador.

Capítulo 3

Potência não é nada sem controle.

Pirelli & C. S.p.A.

SERVIDORES PARA SISTEMAS MULTIPROCESSADOS

Servidores foram concebidos originalmente para estabelecer reservas de tempo a serem utilizadas por tarefas aperiódicas não críticas que competem por processadores com tarefas periódicas críticas, fornecendo a elas uma prioridade maior do que a simples execução em *background*. A Seção 2.2.2 forneceu uma visão geral de diversos tipos de servidores, ilustrando como é possível controlar a interferência exercida por um conjunto de tarefas sobre as demais, isolando-as temporalmente.

Neste Capítulo serão apresentados novos servidores que, ao invés de serem utilizados para isolar a execução de tarefas aperiódicas não críticas, regulam a execução de tarefas críticas em múltiplos processadores, proporcionando as condições adequadas aos algoritmos de escalonamento que serão apresentados nos Capítulos 4 e 5.

Inicialmente será apresentado na Seção 3.1 um conjunto de servidores denominados por *Servidores de Bloqueio Regular* (ou servidores BR). O primeiro destes servidores atende a uma tarefa cliente que necessita executar de forma estritamente periódica em um único processador, bloqueando o uso deste processador por outras tarefas em intervalos regulares. Os outros servidores do conjunto de servidores BR atendem cada um a uma das demais tarefas que compartilham este mesmo processador, garantindo que todas as tarefas consigam cumprir os seus *deadlines*. Posteriormente será apresentado na Seção 3.2 o servidor denominado por *Servidor de Taxa-Fixa* que garante o atendimento dos *deadlines* dos seus clientes, desde que estes exijam em conjunto uma utilização do processador não superior à taxa de execução do servidor. Assim, um conjunto de tarefas clientes pode passar a ser visto como uma entidade única (um servidor de taxa-fixa) que deve executar a taxa constante entre cada dois *deadlines* consecutivos.

3.1 CONTROLE DA EXECUÇÃO DE UMA TAREFA EM DOIS PROCESSADORES

Algumas das estratégias de escalonamento para ambientes com múltiplos processadores exigem que uma tarefa específica execute exclusivamente em reservas de tempo de tamanho

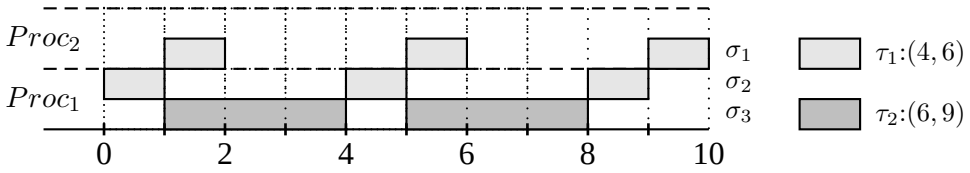


Figura 3.1. Exemplo ilustrativo de tarefas bloqueadas por reservas de tempo regulares

fixo que repetem-se periodicamente. Esta é uma situação que, conforme visto no Capítulo 2, ocorre de forma recorrente nos algoritmos que seguem uma abordagem de escalonamento semi-particionada. Nestes casos, além de garantir que a reserva de tempo seja suficiente para a execução da tarefa determinada, também é preciso garantir que o bloqueio produzido pela reserva de tempo nas demais tarefas não ocasione alguma perda de *deadline*. Portanto, além de construir um mecanismo que restrinja a execução de uma tarefa a intervalos regulares, é preciso que haja um mecanismo que garanta o atendimento dos *deadlines* das demais tarefas eventualmente bloqueadas.

A primeira estratégia de reserva de tempo apresentada neste capítulo é implementada através da combinação de dois tipos de servidores. O primeiro tendo por objetivo o atendimento a uma tarefa que se deseja executar de forma fixa em intervalos regulares de tempo, e o segundo que tem como missão garantir que a tarefa atendida cumpra os seus *deadlines*, mesmo quando sujeita à bloqueios regulares de tamanho fixo. O Exemplo 3.1.1 ilustra a necessidade destes servidores.

Exemplo 3.1.1. *Seja* $\Gamma = \{\tau_1 : (4, 6), \tau_2 : (6, 9)\}$ *um conjunto com duas tarefas periódicas que deverão ser escalonadas em dois processadores* $\Pi = \{Proc_1, Proc_2\}$*. Todas as tarefas liberam jobs no instante* $t = 0$*. A Figura 3.1 mostra um escalonamento em que* τ_1 *é executado em reservas regulares de tempo fornecidas pelos servidores* σ_1 *e* σ_2 *nos dois processadores, enquanto* τ_2 *é atendida por* σ_3 *e executa apenas no primeiro processador. Enquanto* τ_1 *precisa de um servidor que controle o tempo em que ela executa em cada processador,* τ_2 *também não poderia ser simplesmente escalonada por EDF pois, apesar da soma das utilizações no primeiro processador ser igual a* $0.916 < 1$*, qualquer acréscimo no custo computacional de* τ_2 *resultaria na perda do seu deadline em* $t = 9$*.*

3.1.1 Política de Escalonamento

Cada servidor de bloqueio regular (BR) $\sigma{:}(Q, D, T, tipo)$ é caracterizado por uma capacidade de execução Q, um período entre recargas, T, e um *deadline* relativo à sua última recarga, D. O $tipo$ do servidor diferencia os servidores que atendem às tarefas que executam em reservas de tempo regulares, denominados servidores *primários*, dos servidores que atendem às tarefas sujeitas a bloqueios regulares, chamados de servidores *ordinários*. A cada instante, cada servidor possui uma carga, q, $0 \leq q \leq Q$, que é consumida e recarregada durante a execução do sistema de acordo com as regras de consumo e de recarga, respectivamente, determinando o

seu comportamento. Antes de detalhar estas regras, serão definidos os estados dos servidores.

3.1.1.1 Estados dos Servidores

Um servidor pode ser encontrado em um dos quatro estados a seguir:

Pronto. Neste estado, um servidor pode ser escalonado desde que ele possua carga positiva e possua alguma tarefa cliente aguardando execução.

Espera. Não existem tarefas clientes para serem servidas. Portanto, um servidor não pode ser escalonado enquanto estiver neste estado.

Bloqueado. Existe ao menos uma tarefa cliente aguardando execução, mas a carga do servidor já foi toda consumida. Portanto, ele não pode ser escolhido para execução.

Execução. Se um servidor *pronto* for selecionado para execução, ele estará no estado de execução. Em cada instante, existe no máximo um servidor neste estado para cada processador.

As transições entre os estados dos servidores BR são ilustradas na Figura 3.2. Se um servidor σ estiver *pronto*, ele irá para o estado de *execução* assim que for selecionado pelo *dispatcher* para ocupar um processador. Uma vez em *execução*, σ poderá ir para qualquer um dos demais estados a depender da sua carga e da tarefa que ele estiver servindo, conforme indica a figura. Por exemplo, se σ estiver em *execução* e a sua carga terminar antes da conclusão da tarefa cliente servida, ele ficará *bloqueado* e assim permanecerá até que a sua carga seja restabelecida. Por outro lado, se a tarefa servida terminar e não houver uma nova tarefa para ser servida, σ será colocado no estado de *espera* até que um novo job necessite de execução. Se, quando isto ocorrer, a sua carga for positiva ($q > 0$), σ torna-se *pronto*. Caso contrário, ele permanecerá *bloqueado* até que q seja recarregado.

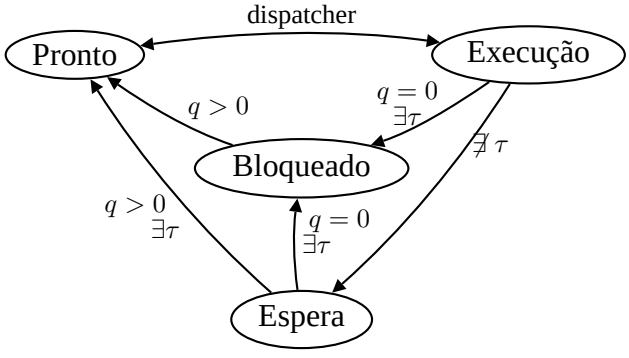


Figura 3.2. Estados dos Servidores BR e suas transições.

3.1.1.2 Regras de Manutenção e Escalonamento dos Servidores

Existem quatro tipos de regras que orientam o comportamento dos servidores BR e o seu escalonamento.

i) **Regras de Recarga**. Quando recarregada, a carga q de um servidor retorna o seu valor à capacidade Q do servidor. Os instantes em que ocorre a recarga dependem dos tipos de servidores:

   a. A carga de um servidor do tipo ordinário é recarregada na liberação da tarefa cliente que ele está servindo. Portanto, pelo modelo de tarefas adotado, o menor intervalo entre recargas é conhecido.
   b. Servidores primários são recarregados no início de cada período regular em que a sua tarefa cliente deve ser executada, i.e. nos instantes kT onde $k = 0, 1, \ldots$ e T é o intervalo entre dois inícios de execução sucessivas.

ii) **Regras de Consumo**. A carga q de um servidor é consumida de duas formas distintas, dependendo do tipo de servidor. Uma vez sendo consumida, o valor da carga decresce a uma taxa de um por unidade de tempo.

   a. Se o servidor for do tipo ordinário, q será consumido quando o servidor estiver no estado de *execução*. Caso contrário q permanecerá inalterado.
   b. No caso do servidor ser primário, q será consumido sempre que possuir valor positivo, independentemente do estado atual do servidor .

iii) **Regra de Antecipação de Deadline**. Considere t o instante de liberação de uma tarefa τ, atendida por um servidor BR $\sigma{:}(Q, T, D, \text{'ordinário'})$. Se um bloqueio regular a que σ estiver sujeito ocorrer no intervalo $[b_1, b_2]$, onde $b_1 < t + D \leq b_2$, o *deadline* de σ será antecipado para $D = b_1$.

iv) **Regra de Escalonamento**. Os servidores BR são escalonados por EDF. Se um servidor ordinário possuir *deadline* igual ao *deadline* de um servidor primário, o escalonador deve selecionar este último em detrimento do primeiro.

É importante enfatizar que devido às regras de recarga (ia e ib) e de consumo (iia e iib), a execução de um servidor primário acontece sempre em intervalos regulares de tempo. As regras de antecipação de *deadline* (iii) e de escalonamento (iv) garantem que nenhum servidor ordinário poderá interferir na execução de um servidor primário, como pode ser visto no Lema 3.1.1.

Lema 3.1.1. *Servidores BR do tipo ordinários não interferem na execução de servidores BR do tipo primários que estabeleçam bloqueios em sua execução. [Massa e Lima 2010]*

Demonstração. A regra de antecipação de *deadlines* (iii) assegura que os *deadlines* absolutos dos servidores ordinários, não serão menores do que os deadlines absolutos dos servidores primários que estejam ativos provocando bloqueios. Como todos os servidores são escalonados por EDF, sendo os desempates decididos em favor dos servidores primários (regra iv), este lema é verdadeiro. □

Tendo garantido que os servidores BR ordinários não provocam interferência na execução dos servidores primários, o Teorema 3.1.1 estabelece condições para que servidores primários e ordinários possam compartilhar um mesmo processador atendendo aos respectivos *deadlines* das suas tarefas clientes.

Teorema 3.1.1. *Um servidor BR primário* $\sigma_p{:}(Q_p, T_p, D_p = Q_p)$ *e um conjunto* Γ *composto por servidores BR ordinários serão escalonáveis por EDF se a Equação 3.1 for satisfeita.*

$$\sum_{\sigma_i(Q_i,T_i,D_i)\in\Gamma} \frac{Q_i}{T_i - Q_p} + \frac{Q_p}{T_p} \leq 1 \tag{3.1}$$

[Massa e Lima 2010]

Demonstração. Decorre do Lema 3.1.1 que servidores primários em execução não sofrem qualquer interferência dos servidores ordinários e portanto executam em mais alta prioridade, cumprindo os seus *deadlines*. Para mostrar que os servidores ordinários também atendem aos seus deadlines é suficiente mostrar que a demanda total dos servidores alocados ao processador em questão não excede 100%, já que o escalonamento é gerado por EDF. Devido às regras de recarga e consumo e ao Lema 3.1.1, o servidor σ_p demanda Q_p/T_p do processador. Como os *deadlines* de todo servidor ordinário σ_i é definido no pior caso pela regra de antecipação do *deadline* (regra iii) como $D_i = T_i - Q_p$, a maior demanda possível para o processador em questão pode ser calculada por

$$\sum_{\sigma_i(Q_i,T_i,D_i)\in\Gamma} \frac{Q_i}{T_i - Q_p} + \frac{Q_p}{T_p}$$

o que, por hipótese, não é maior do que 1 (Equação 3.1).

□

O exemplo 3.1.2 ilustra um cenário particular, mostrando como um conjunto de servidores BR é escalonado.

Exemplo 3.1.2. *Seja* $\sigma_p{:}(1, 1, 3)$ *um servidor BR primário e* $\Gamma = \{\sigma_1{:}(1, 5, 5), \sigma_2{:}(2, 7, 7), \sigma_3{:}(1, 13, 13)\}$ *um conjunto de servidores BR ordinários que deverão ser escalonadas em um mesmo processador. Consideramos que todos os servidores são recarregados inicialmente no instante* $t = 0$*. A Figura 3.3 apresenta o escalonamento produzido para este conjunto de servidores.*

Ao ser realizada a recarga nos servidores BR ordinários, deve-se identificar se o seu deadline corrente ocorrerá em um período em que o servidor sofrerá um bloqueio, ativando a regra de antecipação de deadline. Neste exemplo, isto ocorre com os servidores σ_2 *e* σ_3 *que teriam deadlines respectivamente em* $D_2 = 7 \in [6, 7]$ *e* $D_3 = 13 \in [12, 13]$*, intervalos em que* σ_p *bloqueia a execução dos servidores ordinários. Com isso, os deadlines de* σ_2 *e* σ_3 *são antecipados para* $D_2 = 6$ *e* $D_3 = 12$*. Neste exemplo, o primeiro servidor a ser escalonado é* σ_p*, pois possui o deadline que ocorrerá mais cedo, em* $t = 1$*. Quando a carga de* σ_p *termina*

em $t = 1$, σ_1 começa a ser executado, esgotando a sua carga em $t = 2$. Neste instante, o servidor σ_2 é escalonado e executa até $t = 3$, quando o servidor σ_p é recarregado e retoma a sua execução, provocando uma preempção em σ_2. Com o esgotamento da carga de σ_p em $t = 4$, σ_2 volta a ser escalonado e executa até $t = 5$, atendendo ao seu deadline que havia sido antecipado. O restante do escalonamento prossegue de forma semelhante e todos os servidores cumprem o seu deadline.

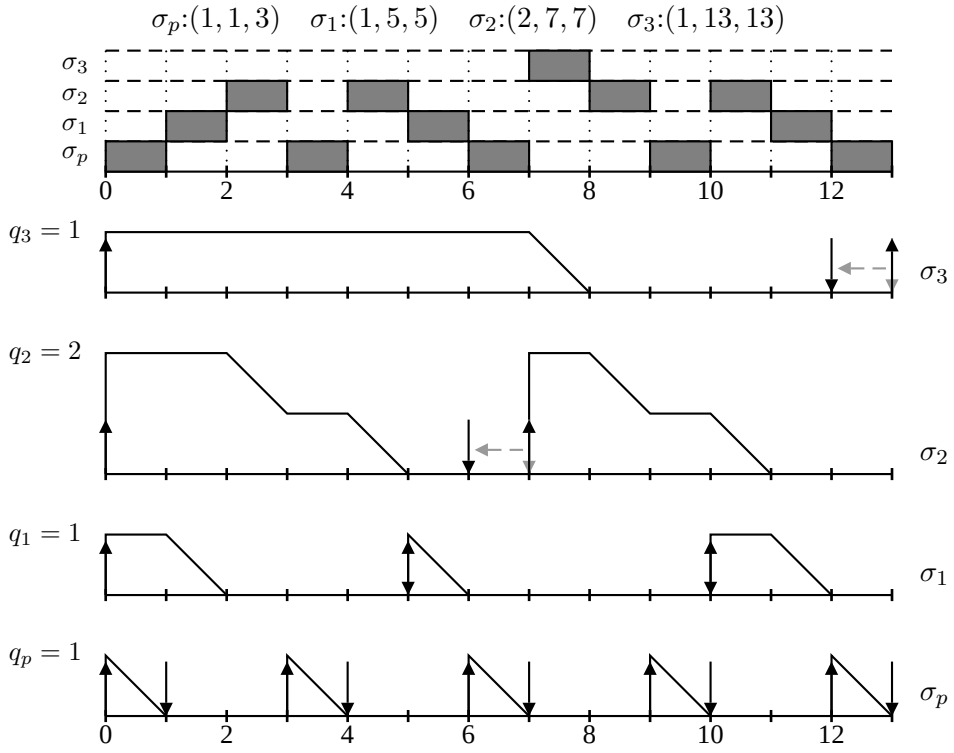
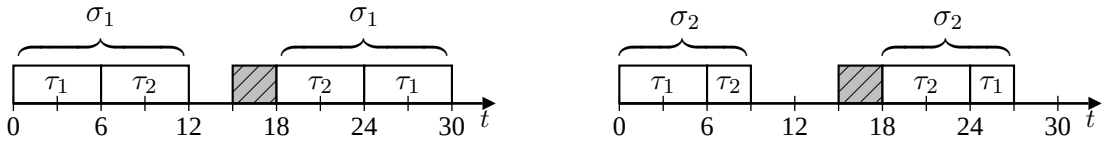


Figura 3.3. Escalonamento produzido para servidores BR.

3.2 CONTROLE DA EXECUÇÃO DE MÚLTIPLAS TAREFAS EM VÁRIOS PROCESSADORES

O segundo mecanismo servidor apresentado neste capítulo, denominado servidor de Taxa-Fixa, é uma generalização do servidor descrito em [Regnier et al. 2012] que tem por objetivo prover adequadamente recursos computacionais a um conjunto de clientes composto por tarefas e por outros servidores de Taxa-Fixa. Este servidor atende a um de seus clientes por vez, permitindo que um conjunto de tarefas seja tratado como uma entidade única que executa a uma taxa constante. Assim, o problema de escalonar um conjunto de tarefas em múltiplos processadores passa a ser reduzido ao problema de escalonar seus respectivos servidores nestes mesmos processadores.



(a) servidor $\sigma_1{:}(0.8, \{\tau_1, \tau_2\})$ (b) servidor $\sigma_2{:}(0.6, \{\tau_1, \tau_2\})$

Figura 3.4. Possíveis escalonamentos produzidos para o Exemplo 3.2.1 por um servidor de Taxa-Fixa $\sigma_1 : (0.8, \{\tau_1, \tau_2\})$ cujos clientes atendem aos seus *deadlines* (3.4a) e por um servidor de Taxa-Fixa $\sigma_2{:}(0.6, \{\tau_1, \tau_2\})$ cujos clientes perdem um *deadline* (3.4b).

3.2.1 Definição dos Servidores de Taxa-Fixa

Definição 3.2.1 (Servidor de Taxa-Fixa)**.** *Um* servidor de taxa-fixa σ *com taxa* $\rho \leq 1$*, é um mecanismo de escalonamento instanciado com o propósito de regular a execução de um conjunto* Γ *composto por tarefas e outros servidores, denotado por* $\sigma{:}(\rho, \Gamma)$*, com* $\mathrm{R}(\sigma) = \rho$*. As regras apresentadas a seguir definem o comportamento deste servidor:*

i) ***Regra de estabelecimento de deadline****. O deadline de um servidor* σ *no instante* t*, denotado por* $D(\sigma, t)$*, nunca será posterior ao próximo deadline previsto para os seus clientes. Esta regra é mais formalmente definida pela Equação 3.2.*

$$D(\sigma, t) \leq \min_{\sigma' \in \mathrm{cli}(\sigma)} \{D(\sigma', t)\}, \forall t \in \mathbb{N} \tag{3.2}$$

ii) ***Regra de liberação de jobs*** *Um servidor* σ *libera um job* $J{:}(c, r, d)$ *com* $c = \mathrm{R}(\sigma)(d - r)$ *e* $d = D(\sigma, r)$ *a todo instante* r *em que algum dos seus clientes também liberar um job.*

iii) ***Regra escalonamento de clientes****. Sempre que um job* J *de um servidor* σ *estiver em execução, na verdade* σ *estará executando um dos seus clientes utilizando EDF.*

Pode-se observar que a definição deste servidor apresenta uma generalização não usual do conceito de tarefa. Assim, pode-se interpretar uma tarefa como um servidor que possua um único cliente ativo a cada instante. Além disso, um servidor de Taxa-Fixa também se comporta de forma semelhante a uma tarefa, liberando jobs cujos tempos de execução são função dos seus *deadlines* relativos ($d - r$).

O Exemplo 3.2.1 ilustra o comportamento de um servidor de Taxa-Fixa.

Exemplo 3.2.1. *Seja* $\Gamma = \{\tau_1{:}(6, 15), \tau_2{:}(12, 30)\}$ *um conjunto de duas tarefas periódicas a serem escalonadas em um único processador.*

A Figura 3.4 apresenta dois possíveis escalonamentos para o Exemplo 3.2.1 produzido pelo servidor $\sigma_1{:}(0.8, \Gamma)$ (3.4a) e pelo servidor $\sigma_2{:}(0.6, \Gamma)$ (3.4b). Em ambos os casos, é assumido que τ_1 e τ_2 são tarefas periódicas, ocorrendo as suas primeiras liberações em $t = 0$. Como pode ser visto na figura, os jobs de σ_1 e σ_2 atendem aos seus *deadlines* já que recebem parcelas de processamento correspondentes às suas taxas de execução. Entretanto, isto não implica que os jobs dos clientes de σ_1 e σ_2 também atendam aos seus *deadlines*.

Considere novamente a Figura 3.4a. Como σ_1 possui *deadline* $D(\sigma_1, 0) = 15$ e o custo computacional do seu primeiro job é $0.8(15-0) = 12$, σ_1 inicialmente executa τ_1 durante $[0, 6)$ e depois seleciona τ_2 para executar em $[6, 12)$. Apesar do job de τ_2 não ter sido totalmente executado, o servidor σ_1 já esgotou a sua carga e portanto não poderá continuar executado até que seja novamente recarregado na liberação do seu próximo job, que ocorrerá no instante $t = 15$. Durante o intervalo $[15, 18)$, outras tarefas/servidores mais prioritárias são escalonadas, como representado na figura. Após este período, σ_1 é escalonado e seus clientes voltam a ser executados cumprindo os seus *deadlines*. Já no cenário ilustrado pela Figura 3.4b, o tempo de processamento disponibilizado por σ_2 para a execução das tarefas em Γ não é suficiente e tanto τ_1 quanto τ_2 perdem os seus *deadlines* durante a execução do segundo job de σ_2. Assim, tanto os jobs de σ_1 quanto os jobs de σ_2 atendem aos seus *deadlines* em $t = 15$ e 30 e, portanto, ambos os cenários retratam escalonamentos válidos para os servidores. Entretanto, devido aos jobs de σ_2 não proverem tempo suficiente para os seus clientes executarem, o escalonamento das tarefas em Γ apresentado na Figura 3.4b não é válido.

No caso particular em que a taxa de execução de um servidor de Taxa-Fixa é igual a soma das taxas de execução dos seus clientes (como apresentado na Figura 3.4a), este servidor comporta-se como o servidor definido por Regnier et al.(2011) e o atendimento dos *deadlines* do servidor implicará no atendimento dos *deadlines* dos seus clientes, como afirma o Teorema 3.2.1.

Teorema 3.2.1. *Um servidor de Taxa-Fixa σ produz um escalonamento válido para os seus clientes em um processador se $\mathrm{R}(\sigma) = \sum_{\sigma' \in \mathrm{cli}(\sigma)} \mathrm{R}(\sigma') \leq 1$ e todos os jobs de σ atenderem aos seus deadlines [Regnier et al. 2011].*

Demonstração. Sem perda da generalidade, pode-se considerar os servidores em $\Gamma = \mathrm{cli}(\sigma)$ como tarefas. Com isso, podem ser aplicados os resultados conhecidos para o escalonamento de sistemas de tarefas. Como não é permitido ao servidor σ executar em mais de um processador ao mesmo tempo, podemos assumir que a sua execução ocorre em um único processador, embora isto não ocorra necessariamente. O teorema será provado inicialmente para o caso em que $\mathrm{R}(\Gamma) = 1$, sendo este resultado posteriormente também estendido para o caso em que $\mathrm{R}(\Gamma) < 1$.

Caso $\mathrm{R}(\Gamma) = 1$: Seja $\eta(t, t')$ a demanda por tempo de processamento no intervalo $[t, t')$, onde $t < t'$. Esta demanda fornece a soma de todos os tempos de execução requisitados por jobs com liberação não anterior a t e com *deadline* não posterior a t'. Por hipótese, esta demanda é limitada por:

$$\eta(t, t') \leq (t' - t) \sum_{\sigma_i \in \Gamma} \mathrm{R}(\sigma_i)$$

e, como por hipótese $\sum_{\sigma_i \in \Gamma} \mathrm{R}(\sigma_i) \leq 1$,

$$\eta(t, t') \leq (t' - t) \tag{3.3}$$

Sabe-se que não existe escalonamento válido para Γ se, e somente se, existir algum intervalo $[t, t')$ em que $\eta(t, t') > t' - t$ [Baruah e Goossens 2004]. A Equação 3.3 garante que isto não ocorre, portanto existe ao menos um escalonamento válido para Γ. Como σ escalona seus clientes por EDF (regra iii), e EDF é ótimo, então σ produz um escalonamento válido para Γ.

Caso $\mathrm{R}(\Gamma) < 1$: Para poder aproveitar o resultado obtido no caso em que $\mathrm{R}(\Gamma) = 1$, será introduzida uma tarefa τ' que preencherá todas as folgas existentes, sendo que τ' libera um job com taxa de execução $\mathrm{R}(\tau') = 1 - \mathrm{R}(\Gamma)$ e com *deadline* $d = D(\Gamma, r)$ a cada instante r em que algum job de uma tarefa em Γ for liberado. Considere $\Gamma' = \Gamma \cup \{\tau'\}$ e um novo servidor de Taxa-Fixa σ' que tenha Γ' como conjunto de clientes e taxa de execução $\mathrm{R}(\sigma') = \mathrm{R}(\Gamma')$. Como $\mathrm{R}(\Gamma') = \mathrm{R}(\Gamma) + \mathrm{R}(\tau') = 1$ então, pelo caso anterior, σ' produz um escalonamento válido para Γ'.

Agora, considere uma janela de escalonamento $[r, d)$, para algum job $J : (c, r, d)$ de σ. Observe que tanto σ quanto τ' liberam jobs com deadline $D = (\Gamma, r)$ sempre que algum job de uma tarefa de Γ for liberado em um instante r. Como σ' produz um escalonamento válido, τ' e σ executam em $[r, d)$ por exatamente $\mathrm{R}(\tau')(d - r)$ e $\mathrm{R}(\sigma)(d - r)$, respectivamente. Como não existem *deadlines* ou novas liberações em $[r, d)$, e contanto que seja preservado o tempo total dedicado à execução de cada cliente neste intervalo, o escalonamento produzido por σ' permanecerá válido mesmo que a execução dos seus clientes seja arbitrariamente rearrumada. Este procedimento pode ser adotado em todas as janelas de escalonamento de forma que qualquer escalonamento gerado por σ seja reproduzido. Finalmente, como σ e σ' utilizam EDF para escalonar as tarefas em Γ, o servidor σ produzirá o mesmo escalonamento válido produzido por σ'.

□

RESUMO

Alguns algoritmos de escalonamento exigem que certas tarefas sejam executadas em reservas de tempo de tamanho fixo que se repetem periodicamente. A execução prioritária destas tarefas nesses períodos provoca bloqueios regulares nas demais tarefas do sistema, dificultando o seu escalonamento. Os servidores BR gerenciam adequadamente este cenário, atendendo às tarefas que executam em intervalos regulares através de servidores do tipo primário e as demais tarefas do sistema através dos servidores do tipo ordinário. Já os servidores de Taxa-Fixa garantem recursos computacionais às suas tarefas cliente, permitindo que seus clientes passem a ser vistos como uma entidade única que executa a uma taxa igual ao somatório das taxas do seu

conjunto de clientes. Nos Capítulos 4 e 5 será visto como estes servidores são utilizados por novas estratégias de escalonamento voltadas para ambientes com múltiplos processadores.

Capítulo

4

Com ordem e com tempo
encontra-se o segredo de fazer tudo
e tudo fazer bem.

Pitágoras

EDF *WITH BANDWIDTH RESERVATION* (EDF-BR)

O algoritmo de escalonamento apresentado neste capítulo, nomeado por EDF-BR (*EDF with Bandwidth Reservation*), segue uma estratégia semi–particionada com migração controlada, similar a alguns dos trabalhos apresentados no capítulo anterior. Um conjunto de tarefas esporádicas independentes, com *deadline* implícito ou arbitrário, é alocado em uma arquitetura com múltiplos processadores considerando-se um teste de aceitação baseado na sua utilização, semelhante ao algoritmo EKG [Andersson e Bletsas 2008] [Andersson e Tovar 2006]. As tarefas são alocadas estaticamente aos processadores e apenas algumas destas têm permissão para migrar entre um par de processadores pré-definidos. A execução das tarefas migratórias é controlada por um sistema de reserva de banda, de forma que a escalonabilidade do sistema seja garantida com utilização do algoritmo de escalonamento EDF.

Para o caso das tarefas com *deadline* arbitrário foi adotado, de forma conservadora, que cada tarefa τ_i termine a sua execução em $\Delta_i = \min(D_i, T_i)$ unidades de tempo após a sua liberação. Para padronizar a notação utilizada, adotaremos também $\Delta_i = D_i = T_i$ para as tarefas com *deadline* implícito. Nesta janela de tempo, a demanda de τ_i pelo processador pode ser definida como $\delta_i = C_i/\Delta_i$. Os resultados obtidos nos experimentos, apresentados na Seção 4.3, indicam um bom desempenho em termos de escalonabilidade quando comparado com os principais trabalhos relacionados propostos recentemente, como *Notional Processors* [Bletsas e Andersson 2009], EDF-WM [Kato et al. 2009], C=D [Burns et al. 2012] e Hime [Santos-Jr. et al. 2013].

4.1 OS SERVIDORES BR NO EDF-BR

Considerando um sistema composto por um conjunto de tarefas esporádicas com *deadline* implícito Γ e uma máquina multiprocessada Π, será descrita uma estratégia de escalonamento

que produzirá um escalonamento válido para este sistema. Esta abordagem é baseada em reservas de banda realizadas em tempo de projeto, sendo que cada reserva é implementada através do escalonamento de um conjunto Ω composto por servidores BR, apresentados no Capítulo 3. Mais especificamente, uma tarefa $\tau_i \in \Gamma$ executa em um processador $Proc_x \in \Pi$ através de um servidor $\sigma_{i,x}$:$(Q_{i,x}, D_{i,x}, T_{i,x}, \text{type})$, significando que τ_i tem o direito de utilizar $Q_{i,x}$ unidades de tempo em cada período de duração $T_{i,x}$ e a sua execução deve ser encerrada antes de $D_{i,x}$, que é o *deadline* do servidor. O valor $Q_{i,x}$ é a capacidade do servidor $\sigma_{i,x}$. O sistema controla a carga atual do servidor $\sigma_{i,x}$, representada por $q_{i,x}$, que pode assumir valores entre 0 e $Q_{i,x}$. Um servidor só pode ser escalonado se a sua carga for não nula. O conjunto de servidores definidos é identificado por Ω, o qual será escalonado de acordo com a política EDF em cada processador $Proc_x \in \Pi$. Em outras palavras, o servidor com o menor *deadline* absoluto no processador $Proc_x$ será escolhido para executar desde que ele possua uma carga positiva e que a sua tarefa associada esteja esperando para ser executada.

Existem três tipos de servidores em Ω. Se houver banda suficiente para executar totalmente uma tarefa τ_i em um processador $Proc_x$, um servidor BR do tipo *ordinário* será definido em Ω, sendo designado para executar em $Proc_x$. Caso contrário, τ_i deverá migrar entre dois processadores, sendo atendido por dois servidores BR do tipo *primário*. Para diferenciá-los neste capítulo, denominaremos estes servidores como *primário* e *secundário*, associados para a execução desta tarefa migratória e alocados em processadores distintos. As regras para a criação de Ω e para a construção do escalonamento destes servidores serão definidas posteriormente neste capítulo.

Motivação e Ilustração

Para ilustrar a estratégia EDF-BR, considere o seguinte exemplo.

Exemplo 4.1.1. *Seja $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ um conjunto de tarefas periódicas que deverão ser escalonadas em dois processadores $\Pi = \{Proc_1, Proc_2\}$. Assuma que $C_1 = 3$, $C_2 = 1.5$, $T_1 = T_2 = 4$, $C_3 = 6$ e $T_3 = 8$, com $D_i = T_i$, $i = 1, 2, 3$. Todas as tarefas são ativadas no instante $t = 0$. A Figura 4.1a mostra um escalonamento feito por EDF Global para este sistema, onde τ_3 não cumpre o seu deadline e $Proc_1$ é subutilizado. No entanto, como pode ser visto na Figura 4.1b, existe um escalonamento válido para este sistema.*

Pode ser observado na Figura 4.1b que o tempo é dividido em janelas com duração igual a $T = 4$ e, em cada janela de tempo, a execução de τ_2 é dividida em duas partes, correspondentes aos servidores $\sigma_{2,1} = (1, 1, 4, \text{‘sec’})$ e $\sigma_{2,2} = (1, 1, 4, \text{‘pri’})$, os quais são escalonados de forma que as suas execuções não se sobreponham no tempo. Enquanto $\sigma_{2,1}$ é responsável pela segunda parte da execução de τ_2, alocada em $Proc_1$ e executando regularmente no final de cada janela de tempo, $\sigma_{2,2}$ cuida da primeira parte da sua execução e executa regularmente no início de cada janela de tempo. Servidores BR somente podem ser escalonados quando suas cargas estiverem

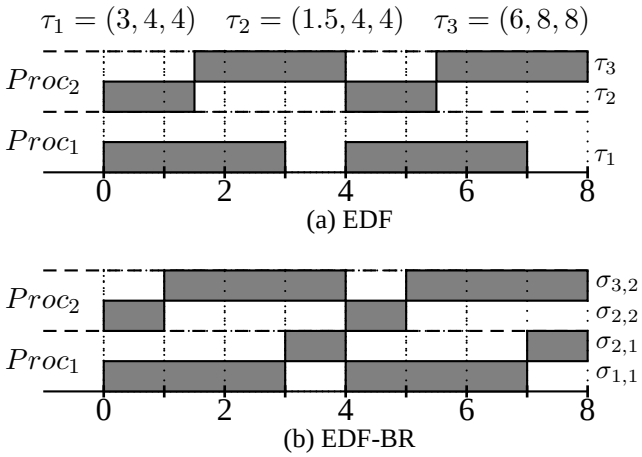


Figura 4.1. Exemplo ilustrativo de (a)EDF e (b)EDF-BR

positivas. Neste exemplo, isto acontece com o servidor primário $\sigma_{2,2}$ nos intervalos $[0, 1)$ e $[4, 5)$, enquanto o servidor secundário correspondente possui carga positiva em $[3, 4)$ e $[7, 8)$.

As tarefas τ_1 e τ_3 são escalonadas através dos servidores ordinários $\sigma_{1,1} = (3, 4, 4,$ 'ord') e $\sigma_{3,2} = (6, 8, 8,$ 'ord'). Na figura em questão, o primeiro servidor possui carga positiva em $[0, 3)$ e $[4, 7)$, enquanto a carga do segundo servidor é positiva durante o intervalo $[0, 8)$. Observe que, pelo Teorema 3.1.1, todos os servidores BR atenderão aos seus respectivos *deadlines*. A seguir, será apresentada uma visão geral sobre o procedimento de alocação de banda.

4.2 A ESTRATÉGIA PROPOSTA

Antes de descrever o procedimento para a criação dos servidores em Ω (Seção 4.2.2), será apresentada uma visão geral na Seção 4.2.1.

4.2.1 Visão Geral

A definição de um conjunto de servidores Ω para atender a um conjunto de tarefas Γ pode ser resumida como se segue:

I- Escolha um tamanho para a janela de tempo $T \leq \Delta_i, i = 1, \ldots, n$ a ser utilizada pelos servidores de todas as tarefas migratórias. Na Figura 4.1b, $T = 4$. Esta abordagem é utilizada para sincronizar as execuções dos servidores em janelas de tempo de forma que as execuções dos servidores associados com a mesma tarefa não executem ao mesmo tempo. Selecione um processador $Proc_x$ para iniciar a alocação dos servidores e vá para o passo 2.

II- Seguindo uma ordem não-crescente de demanda, selecione cada tarefa $\tau_i \in \Gamma$ que ainda não possua um servidor associado. Se for possível, associe uma tarefa a um servidor ordinário no processador corrente $Proc_x$. Caso contrário, vá para o passo 3. Como pode ser visto na Figura 4.1b, neste passo τ_1 e τ_3 foram associados respectivamente a $\sigma_{1,1}$ e $\sigma_{3,2}$.

III- Atribua a maior carga Q ainda disponível no processador corrente $Proc_x$ e selecione a tarefa τ_i que irá migrar.

IV- Crie os servidores secundário e primário, alocados respectivamente aos processadores $Proc_x$ e $Proc_{x+1}$. O primeiro servidor terá capacidade $Q_{i,x} = Q$, enquanto que a capacidade do segundo será $Q_{i,x}$, de forma que τ_i seja atendido totalmente por estes dois servidores. Se isto não for possível, esta abordagem falhará em encontrar um escalonamento factível para o sistema de tarefas em questão. O caso de τ_2 na Figura 4.1b ilustra a forma com que dois servidores associados a uma mesma tarefa são definidos. O servidor secundário utiliza os 25% restantes de $Proc_1$ e o servidor primário é suficiente para completar a execução de τ_2.

V- Tome o próximo processador e repita os passos II-V, até que não reste mais tarefas a serem atendidas, ou não existam mais processadores disponíveis para serem utilizados. No último caso, a estratégia proposta não terá conseguido encontrar um escalonamento válido.

A próxima seção detalhará o esquema de alocação das tarefas pelos processadores e a criação dos servidores BR que controlarão a sua execução.

4.2.2 Alocação de Banda em Processadores

O Algoritmo 4.1 é, na verdade, uma versão detalhada da estratégia de alocação previamente delineada. Todos os servidores primários e secundários são criados com períodos iguais a T, um parâmetro de entrada do algoritmo, que define as *janelas de tempo* para a sincronização da execução dos servidores. Se uma tarefa τ_i precisa migrar entre dois processadores, ela será servida por um par de servidores, um primário e outro secundário. Como τ_i precisa encerrar a sua execução em Δ_i e todos os servidores primários e secundários possuem período T, τ_i precisa terminar em $T\lfloor \Delta_i/T \rfloor \leq \Delta_i$, e irá executar

$$Q_i = \frac{C_i}{\lfloor \Delta_i/T \rfloor} \tag{4.1}$$

em cada janela de tempo. O valor de Q_i é calculado na linha 1 para todas as tarefas, e é utilizado posteriormente na definição das capacidades dos servidores primário e secundário.

Vale ressaltar que o procedimento de alocação de banda tenta preencher completamente um processador com tarefas não-migratórias. Quando isto não for possível, serão criados um servidor primário e um servidor secundário que servirão à tarefa que irá migrar entre o processador corrente e o próximo processador.

Quando um servidor ordinário $\sigma_{i,x}$ for alocado ao processador $Proc_x$, já poderá existir previamente um servidor primário em $Proc_x$, mas nunca um servidor secundário. Denote a capacidade deste servidor primário por Q_x^p, com $0 \leq Q_x^p < \Delta_i$. Deve-se lembrar que servidores

Algorithm 4.1: Procedimento para Alocação de Servidores

Entrada: Conjunto Γ com n tarefas
Entrada: Conjunto Π com m processadores
Entrada: $T \in (0, \min(T_1, \ldots, T_n, D_1, \ldots, D_n)]$
Saída: Se escalonável, Ω; caso contrário, $\emptyset$

1 $Q_i \leftarrow C_i / \lfloor \Delta_i / T \rfloor$, para todo $\tau_i \in \Gamma$
2 Ordene Γ de forma que $\frac{C_i}{\Delta_i} \geq \frac{C_{i+1}}{\Delta_{i+1}}, i = 1, \ldots, n-1$
3 $Q_x^p \leftarrow 0$; $Q_x^s \leftarrow 0, x = 1, \ldots, m$
4 $x \leftarrow 1$; $\delta \leftarrow 0$; $\Omega \leftarrow \emptyset$
5 **while** $x \leq m \wedge \Gamma \neq \emptyset$ **do**
6 **for** $\tau_i \in \Gamma$ **do**
7 $\delta_i' \leftarrow C_i / (\Delta_i - Q_x^p)$
8 **if** $\delta_i' \leq 1 - (Q_x^p / T + \delta)$ **then**
9 $\Gamma \leftarrow \Gamma \backslash \{\tau_i\}$
10 $\Omega \leftarrow \Omega \cup \{S_{i,x} = (C_i, \Delta_i, \Delta_i, \text{`ord'})\}$
11 $\delta \leftarrow \delta + \delta_i'$
12 **if** $x = m \wedge \Gamma \neq \emptyset$ **then**
13 **return** $\emptyset$
14 $Q_x^s \leftarrow \arg\max_{Q \in \mathbb{R}} \{\frac{Q + Q_x^p}{T} + \sum_{S_{i,x} \in \Omega} \frac{Q_{i,x}}{\Delta_i - (Q + Q_x^p)} \leq 1\}$
15 $\tau_i \leftarrow \arg\min_{\tau_j \in \Gamma} \{\frac{Q_j}{T} - \frac{C_j}{\Delta_j} | Q_j \leq T\}$
16 **if** τ_i *foi encontrado* **then**
17 $\Omega \leftarrow \Omega \cup \{S_{i,x} = (Q_x^s, Q_x^s, T, \text{`sec'})\}$
18 $Q_{x+1}^p \leftarrow Q_i - Q_x^s$ $\Omega \leftarrow \Omega \cup \{S_{i,x+1} = (Q_{x+1}^p, Q_{x+1}^p, T, \text{`pri'})\}$
19 $\Gamma \leftarrow \Gamma \backslash \{\tau_i\}$
20 $x \leftarrow x + 1$; $\delta \leftarrow 0$
21 **if** $\Gamma \neq \emptyset$ **then**
22 **return** $\emptyset$
23 **else**
24 **return** Ω

BR do tipo primário devem executar sempre no início de uma janela de tempo de tamanho T e, além disso, a sua execução não sofre interferência pela execução de servidores BR do tipo ordinário devido à regra (iii) de antecipação do *deadline*, apresentada na Seção 3.1.1. Como, no momento da alocação dos servidores do tipo ordinário, a quantidade de tempo que um *deadline* é encurtado está limitada apenas pela capacidade de um servidor primário, a demanda dos servidores ordinários pode ser aumentada para $\delta_i' = C_i / (\Delta_i - Q_x^p)$ (linha 7). Note que, no instante em que os servidores ordinários estão sendo alocados em $Proc_x$, nenhum servidor secundário ainda foi atribuído a $Proc_x$. Se houver capacidade suficiente no processador corrente para suprir a demanda δ_i', um servidor ordinário será criado para servir a esta tarefa com $Q_{i,x} = C_i$ e $D_{i,x} = T_{i,x} = \Delta_i$ (linhas 8-10). A variável δ contabiliza a soma das demandas de todos os servidores já alocados em $Proc_x$. Durante o procedimento de alocação de banda, o deadline

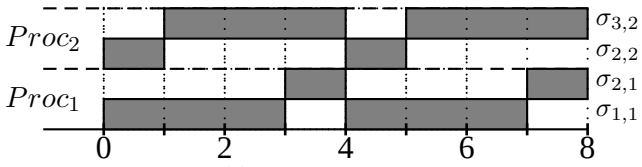
dos servidores é Δ_i. O deadline absoluto dos servidores poderá ser antecipado em tempo de execução, mas apenas se isto for necessário conforme foi explicado na Seção 3.1.1.

Quando não for mais possível alocar servidores ordinários em um processador $Proc_x$, a capacidade restante neste processador poderá ser utilizada na definição de um novo servidor secundário. Neste caso, será necessário mais um processador ($Proc_{x+1}$) já que um novo servidor primário também precisará ser definido. Logo, se $x = m$ e ainda existirem tarefas que ainda não estiverem alocadas em servidores, o algoritmo terá falhado na definição do conjunto de servidores Ω (linha 12). Caso contrário, o algoritmo fará a definição dos servidores primário e secundário.

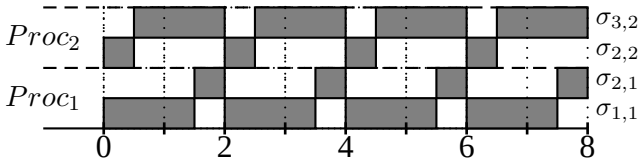
Para aproveitar ao máximo a capacidade de processamento restante em $Proc_x$, o algoritmo procura definir um servidor secundário, e procura por uma tarefa $\tau_i \in \Gamma$ que ainda não tenha sido atendida por nenhum servidor. Isto é feito nas linhas 14-15. Existe um aspecto que vale a pena ser ressaltado nestas linhas. Como um servidor secundário será definido em $Proc_x$, é necessário contabilizar o efeito da antecipação dos deadlines, conforme já mencionado. Agora, os bloqueios produzidos pelas execuções de ambos os servidores primário e secundário devem ser consideradas, e o deadline dos servidores ordinários poderão ser reduzidos em, no máximo, a soma das capacidades dos servidores primário e secundário. Como pode ser visto, se o tempo necessário para a execução do servidor em cada janela de tempo for superior à duração definida para a janela de tempo, é melhor que esta tarefa seja alocada em um servidor ordinário do que ser forçada a migrar. Se alguma tarefa τ_i for escolhida para migrar, um servidor secundário $\sigma_{i,x}$ será criado, com $Q_{i,x} = D_{i,x} = Q^s_x$ e $T_{i,x} = T$, preenchendo totalmente o processador corrente. O seu servidor primário $\sigma_{i,x+1}$, com $Q_{i,x+1} = D_{i,x+1} = Q^p_{x+1}$ e $T_{i,x+1} = T$ será definido para completar o custo total de execução de τ_i (Q_i) na janela de tempo T (linhas 16-18).

É interessante observar que o valor escolhido para T provoca alguns impactos na escalonabilidade do sistema e na quantidade de preempções realizadas. Considerando o exemplo 4.1.1 como ilustração, se fosse escolhido $T = 3$ ao invés de $T = 4$, não seria possível a obtenção de um escalonamento factível. Neste caso, o Algoritmo 4.1 geraria $\sigma_{1,1} = (3, 4, 4, \text{‘ord’})$, $\sigma_{2,1} = (0.458, 0.458, 3, \text{‘sec’})$, $\sigma_{2,2} = (1.542, 1.542, 3, \text{‘pri’})$ e nenhum servidor poderia ser gerado para τ_3. Pode-se perceber que o aumento da demanda do servidor ordinário $\sigma_{1,1}$ em $Proc_1$ seria $\frac{3}{4-0.458} = 0.847$, a demanda do servidor secundário $\sigma_{1,2}$ em $Proc_1$ seria igual a $\frac{0.458}{3} = 0.153$ e a demanda do servidor primário $\sigma_{2,2}$ em $Proc_2$ seria $\frac{1.542}{3} = 0.514$. A demanda aumentada do servidor $\sigma_{3,2}$, bloqueada pela carga do servidor primário $Q^p_2 = 1.542$, seria $\frac{6}{8-1.542} = 0.929$, superior ao que estava disponível em $Proc_2$.

Note ainda que, quanto menor for o valor de T, maior será a quantidade de preempções sofridas pelas tarefas envolvidas. A Figura 4.2 compara o escalonamento produzido para o sistema apresentado no Exemplo 4.1.1 através de EDF-BR utilizando uma janela de tempo $T = 4$ (Figura 4.2a) com EDF-BR utilizando uma janela de tempo $T = 2$ (Figura 4.2b). Pode ser observado que, com uma janela de tempo menor ($T = 2$), a duração das execuções dos servidores primários e secundários também é menor, no entanto estes servidores executam com



(a) EDF-BR com $T = 4$



(b) EDF-BR com $T = 2$

Figura 4.2. Interferência da janela de tempo utilizada pelo EDF-BR na quantidade de preempções com (a) janela de Tempo $T = 4$ e (b) janela de tempo $T = 2$

uma maior frequência, preemptando os servidores ordinários um maior número de vezes.

Estas observações sugerem que se T assumir o maior valor em que $T_i - T\lfloor \Delta_i/T \rfloor$ seja minimizado, pode-se obter um bom equilíbrio entre escalonabilidade e quantidade de preempções, um aspecto também observado em outras abordagens [Andersson e Bletsas 2008] [Andersson e Tovar 2006]. Claramente, escolhendo $T = \text{mdc}(\Delta_1, \ldots, \Delta_n)$ torna $T_i - T\lfloor \Delta_i/T \rfloor = 0$ para toda τ_i. Todavia, se os períodos das tarefas forem co-primos, esta estratégia implicará em alto custo com preempções.

O exemplo a seguir ilustra mais detalhadamente um cenário particular, mostrando como o conjunto de tarefas apresentado no Exemplo 4.1.1 é escalonado pelos servidores BR segundo o algoritmo EDF-BR.

Exemplo 4.2.1. *Considere o conjunto de tarefas apresentado no Exemplo 4.1.1. Os primeiros jobs das tarefas τ_1, τ_2 e τ_3 chegam nos instantes 3, 2 e 1, respectivamente, conforme está indicado pelas setas para cima na Figura 4.3. A partir de então, os demais jobs chegam periodicamente. Os deadlines dos jobs estão representados pelas setas para baixo. A figura apresenta a alteração da carga dos servidores ao longo do tempo. O tamanho da janela de tempo escolhido é $T = 4$.*

Como pode ser observado neste exemplo, $Proc_1$ e $Proc_2$ estão ociosos até os instantes 3 e 1, respectivamente. No entanto, a carga $q_{2,2}$, pertencente ao servidor primário $\sigma_{2,2}$, é consumida desde o instante da sua recarga, conforme a regra de consumo dos servidores BR (regra iib) apresentada na Seção 3.1.1.2. Quando τ_2 chega, no instante $t = 2$, esta carga é nula e ele deve esperar até que seus servidores sejam recarregados. Por outro lado, as cargas dos servidores ordinários são mantidas inalteradas enquanto os servidores não estão executando, de acordo com a regra de consumo (regra iia), também apresentada na Seção 3.1.1.2. A tarefa τ_2 começa a ser servida no instante $t = 3$ por $\sigma_{2,1}$, um servidor secundário. A tarefa τ_1 não é servida na primeira janela de tempo pois $\sigma_{1,1}$ sofreu preempção por $\sigma_{2,1}$ até $t = 4$. Neste instante, o servidor primário foi recarregado. Assim, $\sigma_{1,1}$ e $\sigma_{2,2}$ iniciam a sua execução já que são os

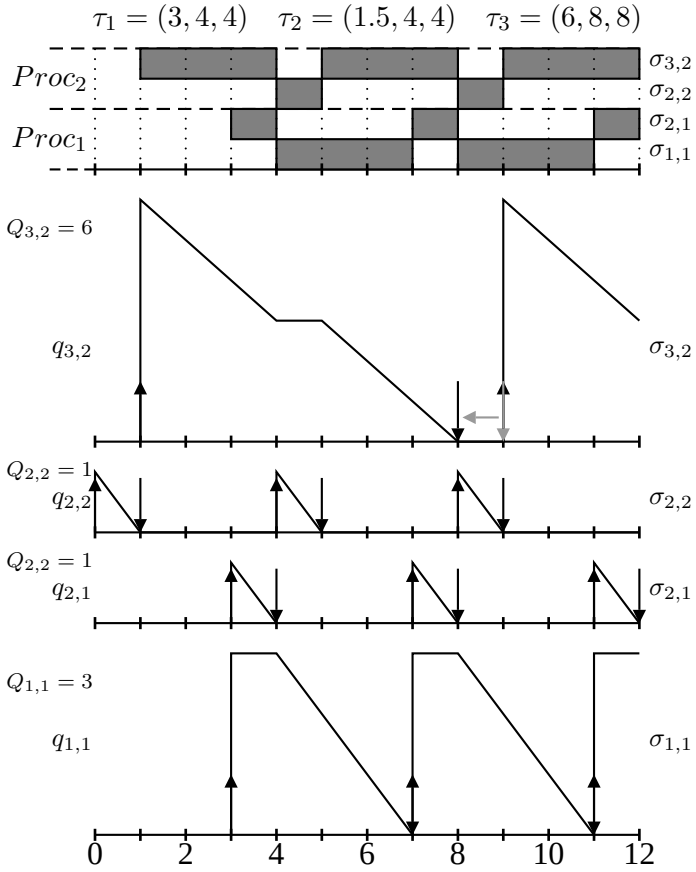


Figura 4.3. Um escalonamento factível produzido pelos servidores BR segundo o algoritmo EDF-BR.

servidores que estão no estado de *pronto* e que que possuem o menor deadline absoluto em $Proc_1$ e $Proc_2$, respectivamente. Pode-se observar também que o servidor $\sigma_{3,2}$ teve o seu *deadline* antecipado de $t = 9$ para $t = 8$ devido a regra de antecipação de *deadline* (regra iii). Como pode ser visto, os servidores BR criados fornecem tempo de processamento suficiente para que todas as tarefas cumpram os seus *deadlines*.

4.3 ANÁLISE DE ESCALONABILIDADE DE EDF-BR

Nesta seção será mostrado que a abordagem EDF-BR encontra um escalonamento factível para um sistema, sempre que o Algoritmo 4.1 terminar de forma bem sucedida. A partir deste ponto será assumido que Ω é o conjunto de servidores definido pelo Algoritmo 4.1, T é a janela de tempo utilizada na definição de Ω para algum conjunto de tarefas Γ e um multiprocessador Π. Os servidores em Ω são servidores BR e seguem as regras estabelecidas na Seção 3.1. Inicialmente será apresentado o Lema 4.3.1, que estabelece que a execução dos servidores primários e secundários não sofrem interferência de nenhum outro servidor.

Lema 4.3.1. *A execução de um servidor primário e de um servidor secundário associados ao mesmo processador ou à mesma tarefa não se sobrepõem no tempo.*

Demonstração. Se não existir um servidor primário ou secundário em Ω, o lema é verdadeiro

de forma trivial. Caso contrário, pelo fato destes servidores serem gerados aos pares pelo Algoritmo 4.1, deve existir ao menos um par de servidores em Ω. Sejam σ^p e σ^s um par qualquer de servidores BR primário e secundário com cargas Q^p e Q^s, respectivamente. σ^p e σ^s são recarregados nos instantes kT e $(k+1)T - Q^s, \forall k \in \mathbb{N}$, de acordo com a regra de recarga de servidores BR (regra ib), apresentada na Seção 3.1.1.2. Pelas regras de consumo iia e iib, também apresentadas na Seção 3.1.1.2, as suas cargas se extinguem nos instantes $kT + Q^p$ e kT, respectivamente. Deste modo, σ^p e σ^s só podem ser escalonados para executarem no início e no final de cada janela de tempo, e as suas execuções não se sobrepõem em janelas de tempo diferentes. Assuma, por contradição, que as execuções se sobrepõem em uma mesma janela de tempo. Pelas regras de consumo e recarga, isto significa que $kT + Q^p > (k+1)T - Q^s$, o que implica que

$$Q^p + Q^s > T \tag{4.2}$$

Existem dois casos a serem considerados, ou σ^p e σ^s servem a mesma tarefa τ_i, ou eles estão alocados no mesmo processador, digamos $Proc_x$. Para o primeiro caso, da linha 18 do Algoritmo 4.1 tem-se que $Q_i = Q^s + Q^p$ e pela linha 15 temos que $Q_i \leq T$, o que contradiz a Equação (4.2).

Agora considere o caso em que σ^p e σ^s são alocados em um mesmo processador $Proc_x$. Q^s é calculado pela linha 14 do Algoritmo 4.1 de forma que

$$\frac{Q^s + Q^p}{T} + \underbrace{\sum_{\sigma_{i,x} \in \Omega} \frac{Q_{i,x}}{\Delta_i - (Q^s + Q^p)}}_{\geq 0} \leq 1 \tag{4.3}$$

Como $\Delta_i > (Q^s + Q^p)$, a Equação (4.3) implica que $Q^s + Q^p \leq T$, o que também contradiz (4.2). □

Pode ser mostrado agora que todos os servidores em Ω cumprem o seu *deadline*.

Lema 4.3.2. *O conjunto de servidores Ω é escalonável por EDF.*

Demonstração. Decorre dos Lemas 4.3.1 e 3.1.1 que servidores primários e secundários em execução não sofrem qualquer interferência e atendem aos seus deadlines. Além disso, como o escalonamento gerado segue a ordem EDF, para mostrar que os servidores ordinários também atendem aos seus deadlines é suficiente mostrar que a demanda dos servidores alocados ao processador $Proc_x$ não excede 100%. Considere σ^p e σ^s os servidores primário e secundário alocados ao processador $Proc_x$ e assuma que as suas capacidades são Q^p e Q^s, respectivamente. Devido às regras de reabastecimento e consumo, e aos Lemas 4.3.1 e 3.1.1, estes servidores demandam $(Q^p + Q^s)/T$ do processador $Proc_x$. Como os deadlines dos servidores ordinários $\sigma_{i,x}$ são definidos no pior caso como $\Delta_i - (Q^s + Q^p)$, a maior demanda em $Proc_x$ é dada por

$$\frac{Q^s + Q^p}{T} + \sum_{\sigma_{i,x} \in \Omega} \frac{Q_{i,x}}{\Delta_i - (Q^s + Q^p)},$$

o que não é maior do que 1 pelo Algoritmo 4.1 (linha 14). □

Agora, será mostrado que se o Algoritmo 4.1 gerar Ω para um sistema de tarefas, este será escalonável pela abordagem proposta.

Teorema 4.3.1. *Seja Ω um conjunto de servidores gerados pelo Algoritmo 4.1 para uma dada janela de tempo e um dado conjunto de tarefas Γ. Nenhuma tarefa deixará de atender ao seu deadline se o(s) seu(s) servidor(es) associado(s) em Ω for(em) escalonado(s) por EDF.*

Demonstração. Sem perda de generalidade, considere um job de alguma tarefa $\tau_i \in \Gamma$ liberado no instante t. Relembre que foi requerido que qualquer tarefa τ_i terminasse a sua execução no intervalo $\Delta_i = \min(D_i, T_i)$. Como nenhum outro job desta tarefa estará sendo considerado, este job será referenciado simplesmente como τ_i para simplificar a notação.

Dois casos devem ser considerados. Assuma, inicialmente, que τ_i é servida pelo servidor ordinário σ alocado a algum processador $Proc_x \in \Pi$. Note que a carga de σ é igual a C_i, σ cumpre o seu deadline (pelo Lema 4.3.2) e $t + \Delta_i$ não é inferior ao deadline de σ. Logo, é garantido que τ_i terminará a sua execução cumprindo o seu deadline.

Agora, considere que τ_i é servido por um par de servidores primário e secundário, σ^p e σ^s, respectivamente. Pelo Lema 4.3.2, há $\eta_i = \lfloor \Delta_i / T \rfloor$ instâncias destes servidores que atendem aos seus *deadlines*. Além disso, o *deadline* da última instância destes servidores não é superior a $t + \Delta_i$. Ainda, pelo Lema 4.3.1, não há sobreposição entre as execuções de σ^p e σ^s. Portanto eles servem $\eta_i(Q^p + Q^s) = C_i$ (pela linha 1 do Algoritmo 4.1) unidades de tempo em Δ_i, o que é suficiente para concluir τ_i com êxito. Logo, τ_i também não poderá perder o seu *deadline* no segundo caso. □

4.4 AVALIAÇÃO DO EDF-BR

Apesar do EDF-BR também poder ser aplicado em sistemas com tarefas esporádicas de *deadline* arbitrário, para efeito de sua avaliação foram utilizados sistemas com *deadline* implícito de forma a poder compará-lo com trabalhos relacionados que somente atendem a este tipo de tarefa.

Foram gerados conjuntos de tarefas de acordo com o procedimento descrito por Emberson *et al.* [Emberson et al. 2010]. As taxas de utilização das tarefas foram produzidas no intervalo $(0, 1)$ e seus períodos inteiros foram gerados em $[1, 100]$, todos distribuídos uniformemente. Os sistemas de tarefas foram gerados com utilização total no intervalo $[0.80, 0.99]$.

Os resultados dos experimentos realizados serão apresentados na Seção 4.4.1, sendo o desempenho do EDF-BR comparado com o desempenho obtido por trabalhos relacionados. A maior parte dos resultados será apresentada através de gráficos, sendo cada ponto em um gráfico corresponde a aproximadamente 1.000 conjuntos de tarefas avaliados, totalizando uma média de 20.000 conjuntos de tarefas por gráfico construído. A métrica de comparação utilizada foi o percentual de sistemas escalonados em cada conjunto. Assim, algoritmos ótimos que conseguem

escalonar 100% dos sistemas com utilização total não superior à quantidade de processadores do sistema não foram considerados. Assim, os algoritmos U-EDF [Nelissen et al. 2012] e QPS (Capítulo 5) não foram considerados nestes experimentos.

4.4.1 Resultados Comparativos

A concepção do algoritmo EDF-BR data de 2010 [Massa e Lima 2010]. Após a sua publicação, surgiram novos algoritmos de escalonamento relacionados que foram inseridos nesta seção de experimentos para que a avaliação do EDF-BR com o atual estado da arte seja a mais atualizada possível. EDF-BR foi avaliado através da sua comparação com os algoritmos EDF-WM [Kato et al. 2009] e *Notional Processors* [Bletsas e Andersson 2009], propostos anteriormente à publicação do EDF-BR em 2010, e com os algoritmos C=D [Burns et al. 2012] e Hime [Santos-Jr. et al. 2013], apresentados em publicações mais recentes.

Assim como o EDF-BR, o *Notional Processors* faz uso de um parâmetro referente a uma *janela de tempo* T, que é definida como o menor dos períodos divididos por um fator δ. Nos experimentos realizados, foi atribuído a este fator o valor 4 para a configuração do EDF-BR, conforme recomendado por Andersson e Bletsas(2008) e utilizado em outras avaliações experimentais [Kato et al. 2009]. Para o *Notional Processors*, foram utilizadas as versões que utilizam $\delta = 1$ e 4.

As Figuras 4.4, 4.5 e 4.6 apresentam os resultados obtidos em experimentos com respectivamente 4, 8 e 16 processadores. Em cada figura são apresentados três casos: (a) o caso em que o sistema é composto por $m + 1$ tarefas, reduzindo a possibilidade de um escalonamento particionado; (b) o caso em que o sistema é composto por $2m$ tarefas; e (c) o caso em que o sistema é composto por $3m$ tarefas, aumentando consideravelmente a possibilidade de particionamento.

Na Figura 4.4 tem-se os resultados dos experimentos utilizando sistemas com apenas quatro processadores. Quando a quantidade de tarefas em cada sistema é de cinco tarefas (Figura 4.4a), ou seja, as tarefas possuem utilização mais alta e ocorrem em menor quantidade sem que o sistema seja obrigatoriamente particionado, EDF-BR é o algoritmo que possui o melhor desempenho, sendo este equivalente ao desempenho do algoritmo C=D, desenvolvido dois anos mais tarde. Conforme a quantidade de tarefas aumenta para esta mesma quantidade de processadores (Figuras 4.4b e 4.4c), o desempenho do EDF-BR melhora, assim como o desempenho de todos os demais algoritmos avaliados, mas não consegue acompanhar o aumento do desempenho do algoritmo C=D. É importante observar que a diferença entre EDF-BR e C=D é significativa apenas em sistemas com utilização total superior a 98%.

Quando os sistemas avaliados possuem oito processadores (Figura 4.5), os desempenhos do EDF-BR e do C=D não sofrem alterações significativas. No entanto, o desempenho destes dois algoritmos passam a ter a companhia do desempenho do algoritmo *Notional Processors* [Bletsas e Andersson 2009] com a aplicação do fator $\delta = 4$.

A Figura 4.6 apresenta os desempenhos obtidos nos experimentos com uma maior quan-

tidade de processadores. Quando a quantidade de tarefas do sistema é grande (48 tarefas na Figura 4.6c), o desempenho da maioria dos algoritmos se assemelha em decorrência da maior probabilidade de particionamento do conjunto de tarefas pelos processadores. Para os sistemas com menos tarefas (17 na Figura 4.6a e 32 na Figura 4.6b), o algoritmo C=D se destaca por ter melhor desempenho para sistemas com 99% de utilização. Para os sistemas com até 98% de utilização, os algoritmos *Notional Processors* e EDF-BR apresentam desempenho tão bom quanto o algoritmo C=D.

4.4.2 Análise do Limite Superior de Preempções e de Migrações do EDF-BR

A quantidade de preempções provocadas durante a execução do algoritmo EDF-BR é uma função da duração da janela de tempo adotada. Cada processador tem, a cada intervalo de duração T, a execução de um par de servidores BR primário e secundário. Ao ser recarregado um servidor secundário, ele poderá provocar uma preempção em um servidor ordinário que estiver em execução. Da mesma forma, ao terminar a carga do servidor secundário e ser iniciada a execução do servidor primário, a tarefa cliente do servidor secundário também sofrerá uma preempção. Por fim, ao terminar a carga do servidor primário, o cliente deste também sofrerá uma preempção. Assim, em cada janela de tempo, o início da execução de um servidor BR do tipo secundário e os términos das cargas dos servidores BR primário e secundário provocarão um total de 3 preempções em cada processador, totalizando um máximo de $3\left\lceil\frac{L}{T}\right\rceil$ preempções em um intervalo de duração L em cada processador. É importante notar que $2\left\lceil\frac{L}{T}\right\rceil$ destas preempções ocasionam a migração da tarefa preemptada. Esta é a quantidade de preempções provocadas em tarefas atendidas por servidores primários e secundários e que deverão retomar a sua execução posteriormente em outro processador.

Cada servidor BR do tipo ordinário também pode provocar uma preempção toda vez que for recarregado, já que os servidores são escalonados por EDF. Portanto, tem-se que a quantidade de preempções provocadas por um servidor BR $\sigma_i{:}(Q_i, D_i, T_i,$‘ord’$)$ em um intervalo de duração L não será maior do que $\left\lceil\frac{L}{T_i}\right\rceil$. Considere `Preempt`($L$,Proc) e `Migr`($L$,Proc) respectivamente as quantidades de preempções e de migrações ocorridas em um processador Proc em um intervalo de duração L. Limites superiores para `Preempt`(L,Proc) e `Migr`(L,Proc) podem ser expressos pelas equações 4.4 e 4.5.

$$\texttt{Preempt}(L, \text{Proc}) \leq \sum_{\tau_i \in \text{Proc}} \left\lceil \frac{L}{T_i} \right\rceil + 3 \left\lceil \frac{L}{T} \right\rceil \tag{4.4}$$

$$\texttt{Migr}(L, \text{Proc}) \leq 2 \left\lceil \frac{L}{T} \right\rceil \tag{4.5}$$

Para os casos particulares do primeiro e do último processador, estes limites são reduzidos respectivamente para $\sum_{\tau_i \in \text{Proc}} \left\lceil\frac{L}{T_i}\right\rceil + 2\left\lceil\frac{L}{T}\right\rceil$ e para $\left\lceil\frac{L}{T}\right\rceil$, já que estes processadores não possuem servidores primário e secundário, respectivamente.

Assim, considerando $\texttt{Preempt}(L)$ e $\texttt{Migr}(L)$ respectivamente o total de preempções e de migrações ocorridas em todo o sistema em um intervalo de duração L, temos as Equações 4.6 e 4.7 fornecendo limites superiores para $\texttt{Preempt}(L)$ e $\texttt{Migr}(L)$, onde $\widetilde{\Gamma}$ é o conjunto das tarefas do sistema que não migram.

$$\texttt{Preempt}(L) \leq \sum_{\tau_i \in \widetilde{\Gamma}} \left\lceil \frac{L}{T_i} \right\rceil + (3m-2) \left\lceil \frac{L}{T} \right\rceil \tag{4.6}$$

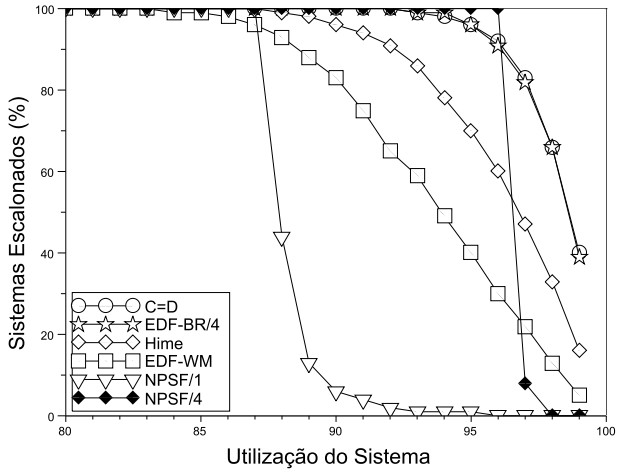
$$\texttt{Migr}(L) \leq 2 \left\lceil \frac{L}{T} \right\rceil \tag{4.7}$$

RESUMO

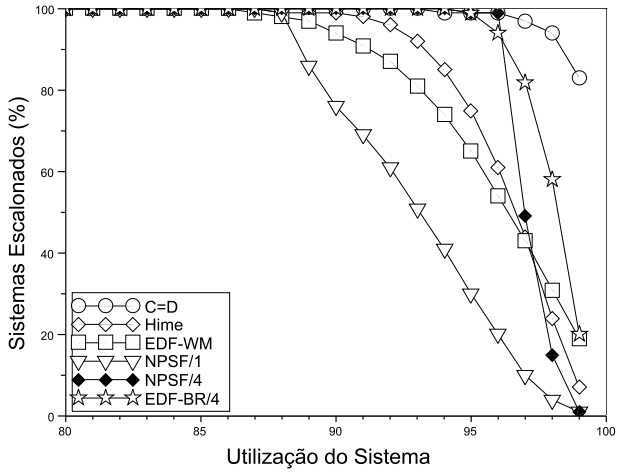
O algoritmo EDF *with Bandwidth Reservation* segue uma abordagem semi-particionada para o escalonamento de tarefas esporádicas em múltiplos processadores. Em um procedimento *off-line*, as tarefas do sistema são distribuídas sequencialmente pelos processadores e, quando não for mais possível a alocação total de nenhuma tarefa no processador atual, é escolhida uma tarefa para migrar entre este e o próximo processador. Como EDF-BR divide o fluxo do tempo em janelas, as tarefas que migram evitam ocupar dois processadores ao mesmo tempo dividindo a sua execução proporcionalmente pelas janelas de tempo e executando em um processador no início, e no outro processador ao final de cada janela. Servidores BR (Seção 3.1) do tipo primário são encarregados da execução das tarefas migratórias, enquanto servidores BR do tipo ordinário atendem às tarefas totalmente alocadas em um único processador.

EDF-BR mostrou-se competitivo quando comparado através de experimentos com os principais trabalhos relacionados. Em particular, como o procedimento descrito para a alocação de tarefas executa em tempo polinomial, EDF-BR é adequado para para sistemas complexos de tarefas executando em arquiteturas com muitos processadores.

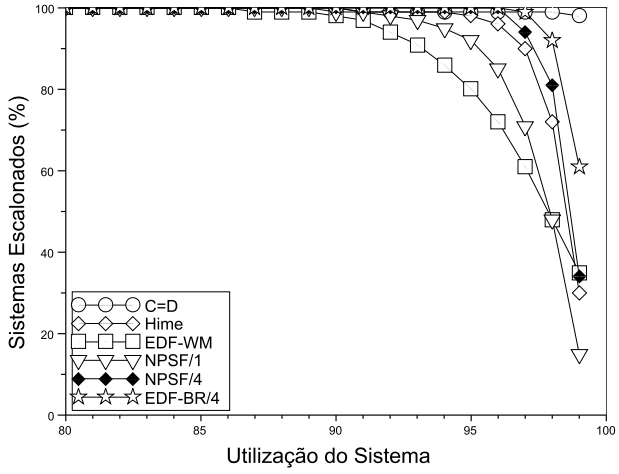
EDF-BR precisa sincronizar o escalonamento em janelas de tempo. Esta característica, também presente em outros algoritmos semi-particionados, pode gerar uma sobrecarga devido ao acesso concorrente ao barramento interno de uma arquitetura multiprocessada. Outra característica observada no EDF-BR é que podem haver sistemas escalonáveis para os quais EDF-BR falha em produzir uma alocação de servidores. Fazendo-se uso de novos esquemas para reserva de banda e de alocação de tarefas, no Capítulo 5 será descrito um novo algoritmo de escalonamento que não sofre destes efeitos colaterais.



(a) $n = 5$ tarefas

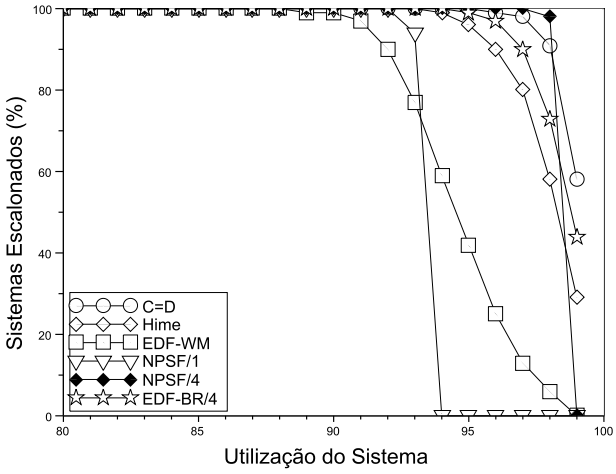


(b) $n = 8$ tarefas

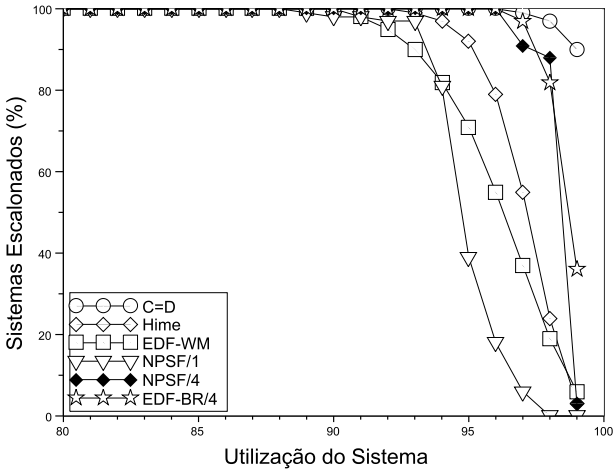


(c) $n = 12$ tarefas

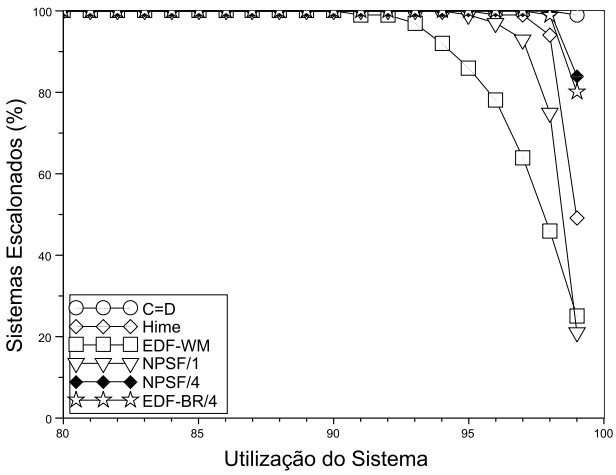
Figura 4.4. Sistemas com n tarefas de *deadline* implícito sendo escalonados em $m = 4$ processadores: (a) $n = 5$; (b) $n = 8$; (c) $n = 12$.



(a) $n = 9$ tarefas

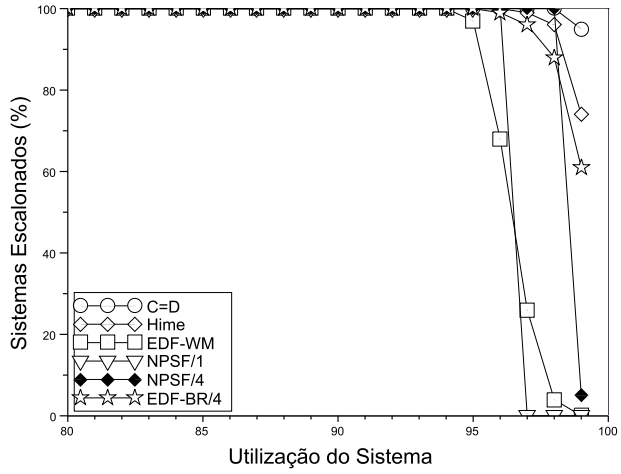


(b) $n = 16$ tarefas

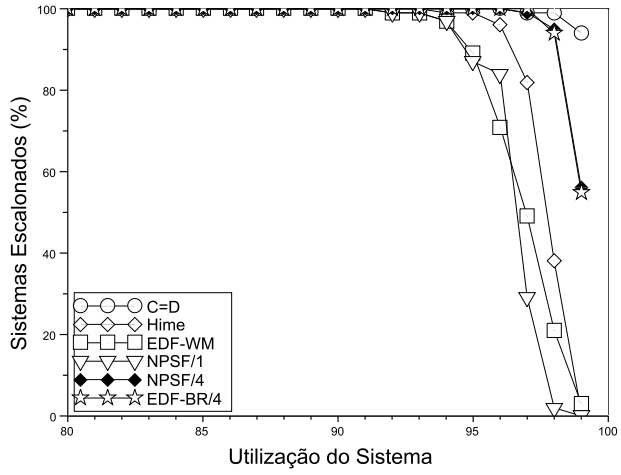


(c) $n = 24$ tarefas

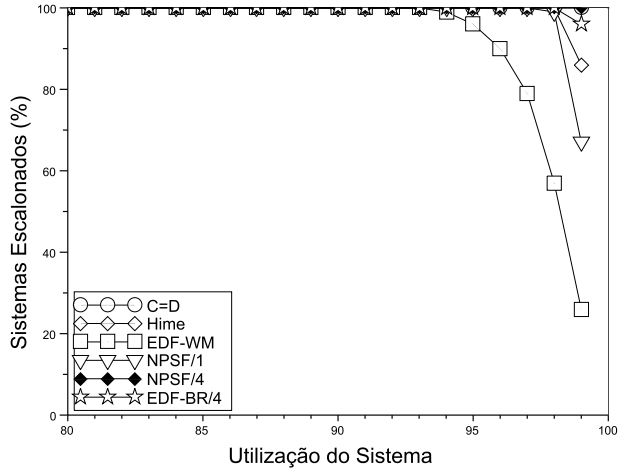
Figura 4.5. Sistemas com n tarefas de *deadline* implícito sendo escalonados em $m = 8$ processadores: (a) $n = 9$; (b) $n = 16$; (c) $n = 24$.



(a) $n = 17$ tarefas



(b) $n = 32$ tarefas



(c) $n = 48$ tarefas

Figura 4.6. Sistemas com n tarefas de *deadline* implícito sendo escalonados em $m = 16$ processadores: (a) $n = 17$; (b) $n = 32$; (c) $n = 48$.

Capítulo

5

O ócio é doce,

mas as suas consequências são cruéis.

John Quincy Adams

ESCALONAMENTO QUASI-PARTICIONADO (QPS)

Conforme visto no Capítulo 2, existem duas abordagens para o escalonamento de sistemas de tempo real em múltiplos processadores que atingem a otimalidade sem forçar a igualdade de *deadlines*, RUN [Regnier et al. 2011, Regnier et al. 2012] e U-EDF [Nelissen et al. 2012]. Apesar de RUN provocar o menor *overhead* devido a preempções e migrações, ele é capaz de lidar apenas com tarefas periódicas. U-EDF consegue escalonar corretamente tarefas esporádicas, entretanto concentra os jobs em poucos processadores provocando preempções mesmo quando ainda existem processadores ociosos. O algoritmo de escalonamento Quasi-Particionado (QPS - *Quasi-Partition Scheduling*), além de lidar corretamente com tarefas esporádicas, adota uma estratégia adaptativa que permite alterar o seu comportamento para um escalonamento particionado quando houver o atraso na chegada de um job.

É importante ressaltar que o quasi-particionamento não deve ser confundido com a estratégia de semi-particionamento adotada pelo EDF-BR (Capítulo 4), além do EKG [Andersson e Tovar 2006, Andersson et al. 2008] e EDF-WM [Kato et al. 2009]. Nestes esquemas, a maior parte das tarefas são associadas a um único processador, enquanto algumas tarefas podem ser escolhidas para migrar entre dois ou mais processadores. Neste aspecto, QPS é mais geral já que conjuntos de tarefas (e não apenas algumas tarefas) recebem a permissão para migrar, sendo que a decisão sobre qual a tarefa que irá migrar é realizada em tempo de execução. Além disso, o semi-particionamento normalmente requer uma escolha entre o alto *overhead* provocado pela sua implementação e a obtenção de uma maior utilização dos processadores, enquanto QPS sempre possibilita a utilização total dos processadores com baixa quantidade de preempções e migrações.

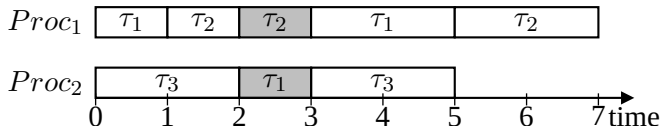


Figura 5.1. Escalonamento QPS para três tarefas. Todas as tarefas chegam no instante 0; tarefas τ_1 e τ_3 são ativadas com período 3; mas o segundo job de τ_2 apenas chega no instante 4. A migração de tarefas ocorre durante o período $[0, 3)$; o escalonamento particionado é aplicado durante $[3, 7)$ devido ao atraso de um job.

5.1 ALGORITMO QPS EM LINHAS GERAIS

QPS é um algoritmo capaz de reunir as vantagens existentes nas abordagens global e particionada, escalonando qualquer sistema factível composto por tarefas esporádicas independentes com *deadline* implícito em um conjunto de processadores idênticos. QPS primeiro particiona o sistema de tarefas em subgrupos, denominados *execution sets*. Quando um *execution set* pode ser escalonado em um único processador ele é denominado *minor execution set*, caso contrário ele é chamado de *major execution set*. Quando todos os *execution sets* são *minor execution sets*, QPS converge para EDF particionado. A depender da demanda corrente, *major execution sets* podem ser escalonados em um único processador por EDF local (*modo EDF*), ou podem ser escalonados por um conjunto de servidores que executam em mais de um processador (*modo QPS*). QPS escolhe o modo de escalonamento a ser utilizado através do monitoramento, em tempo de execução, da demanda de um *major execution set*, proporcionando uma adaptação dinâmica.

A capacidade de adaptação dinâmica do QPS está diretamente relacionada com as propriedades inerentes ao método de particionamento utilizado, chamado *quasi-particionamento*. Para ilustrar esta adaptação, considere um sistema com dois processadores e três tarefas, $\tau_1 : (2, 3)$, τ_2:$(2, 3)$ e τ_3:$(2, 3)$ (Figura 5.1). QPS particiona este conjunto de tarefas em dois sub-conjuntos. Um *major execution set* $P_1 = \{\tau_1, \tau_2\}$ e um *minor execution set* $P_2 = \{\tau_3\}$, que respectivamente requerem $4/3$ e $2/3$ do tempo do processador. P_1 é estaticamente associado a servidores de Taxa-Fixa para garantir que $1/3$ do tempo seja reservado à execução de τ_1 e τ_2 em paralelo (área cinza na Figura 5.1), e por $2/3$ do tempo eles executem de forma alternada em um mesmo processador. Além disso, já que P_2 apenas requer $2/3$ do tempo de um processador, τ_3 pode executar em um único processador. Quando um processador for suficiente para a demanda corrente de P_1 devido ao atraso de alguma instância de τ_1 ou de τ_2, QPS desativa dinamicamente o conjunto de servidores e escalona P_1 utilizando EDF local. Esta adaptação é ilustrada pela Figura 5.1, onde um escalonamento global é aplicado durante $[0, 3)$ enquanto que a chegada no instante 4 de uma instância de τ_2 propicia que P_1 seja escalonado de forma particionada durante $[3, 7)$. QPS é o primeiro algoritmo de escalonamento que propicia este tipo de adaptação em tempo de execução.

5.2 OS SERVIDORES DE TAXA-FIXA NO QPS

A estratégia de escalonamento *Quasi-Partition Scheduling* se utiliza de um conjunto de servidores de Taxa-Fixa (Seção 3.2), construídos de forma adequada para proporcionar um escalonamento adequado. A definição do servidor de Taxa-Fixa apresenta conceitos que são uma generalização do conceito usual de tarefa. Assim, pode-se interpretar uma tarefa como um servidor que possui um único cliente ativo a cada instante. Além disso, um servidor também se comporta de forma semelhante a uma tarefa, libertando jobs cujo tempo de execução é uma função do seu *deadline* relativo ($d - r$). Portanto, neste capítulo, utilizaremos indistintamente o termo "servidor" tanto para servidores como para tarefas; Γ será frequentemente referido como conjunto de servidores.

Inicialmente será apresentada uma visão geral da forma como QPS se utiliza dos servidores de Taxa-Fixa para gerar um escalonamento válido em uma plataforma com múltiplos processadores. Seja Γ um conjunto de servidores que sempre libera um novo cliente a cada *deadline* de qualquer um dos seus jobs. Considere P um conjunto de servidores obtido pelo particionamento de Γ. Quando $\mathrm{R}(P) \leq 1$, P é chamado de *minor execution set* e a simples utilização de um processador dedicado e do escalonamento por EDF garante o atendimento dos requisitos temporais de P. Assuma que $\mathrm{R}(P) = 1 + x$, $0 < x < 1$ e que $x < \mathrm{R}(\sigma)$ para todo servidor σ em P. Neste caso, P é denominado por *major execution set* e requer mais do que um e não mais do que dois processadores para atender aos seus requisitos temporais. Neste caso, representa-se por x a necessidade de execução em paralelo das tarefas em P. Este caso é tratado reservando-se para P um processador inteiro mais uma fração x de um segundo processador. Para tanto, P é biparticionado nos subconjuntos P^A e P^B. Como $x < R(\sigma)$ e $\mathrm{R}(P) < 2$, qualquer biparticionamento de P satisfaz $R(P^A) > x$, $R(P^B) > x$ e $R(P^A) + R(P^B) < 2$. Consequentemente pode-se estabelecer a seguinte definição.

Definição 5.2.1 (Servidores QPS)**.** *Seja P um conjunto de servidores de Taxa-Fixa com taxa acumulada igual a $\mathrm{R}(P) = 1 + x$ tal que $0 < x < 1$. Dado um biparticionamento $\{P^A, P^B\}$ de P, definem-se quatro servidores QPS de Taxa-Fixa associados com P como σ^A:$(\mathrm{R}(P^A) - x, P^A)$, σ^B:$(\mathrm{R}(P^B) - x, P^B)$, σ^S:(x, P) e σ^M:(x, P). Todos os servidores QPS associados com P compartilham um mesmo deadline $D(P, t)$ em qualquer instante t. σ^A e σ^B são servidores dedicados respectivamente a P^A e P^B, e os servidores σ^M e σ^S são denominados servidores master e slave.*

Os servidores σ^A e σ^B são encarregados da execução não paralela de P^A e P^B, respectivamente, enquanto σ^M e σ^S tratam da sua execução paralela. Como $\mathrm{R}(\sigma^A) + \mathrm{R}(\sigma^B) + \mathrm{R}(\sigma^S) = 1$, σ^A, σ^B e σ^S podem executar em um único processador, enquanto σ^M executa em um processador diferente. Sempre que σ^M for selecionado para executar, σ^S também executará, o que justifica as suas denominações. Além disso, sempre que σ^M e σ^S executarem, uma tarefa de P^A e uma tarefa de P^B executarão em paralelo; a escolha de qual será executada por σ^M e qual será executada por σ^S é apenas uma questão de eficiência.

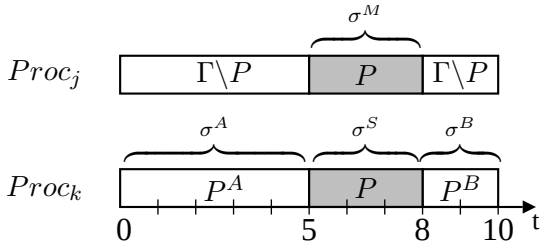


Figura 5.2. Ilustração de um possível escalonamento produzido por QPS para um conjunto Γ de tarefas que utilizam completamente dois processadores. Um subconjunto $P = \{\tau_1 : (6, 15), \tau_2 : (12, 30), \tau_3 : (5, 10)\}$ de Γ é biparticionado em $P^A = \{\tau_1, \tau_2\}$ e $P^B = \{\tau_3\}$. Servidores QPS são definidos como σ^A:$(0.5, P^A)$, σ^B:$(0.2, P^B)$, σ^M:$(0.3, P)$ e σ^S:$(0.3, P)$.

O Exemplo 5.2.1 ilustra como os servidores são utilizados pelo QPS.

Exemplo 5.2.1. *Seja Γ um conjunto de tarefas periódicas que utilizam totalmente dois processadores e considere $P = \{\tau_1$:$(6, 15), \tau_2$:$(12, 30), \tau_3$:$(5, 10)\}$ um subconjunto de Γ.*

Como $\mathrm{R}(P) = 1 + 0.3$, um biparticionamento de P pode ser definido como $P^A = \{\tau_1, \tau_2\}$ e $P^B = \{\tau_3\}$. Os servidores QPS $\sigma^A, \sigma^B, \sigma^M$, e σ^S podem ser definidos como $\sigma^A : (0.5, P^A)$, $\sigma^B : (0.2, P^B)$, $\sigma^M : (0.3, P)$ e $\sigma^S : (0.3, P)$ (Figura 5.2). Servidores σ^A, σ^B e σ^S são alocados ao mesmo processador, enquanto σ^M executa em outro processador. Sempre que σ^M for escalonado para executar (seguindo a ordem EDF em seu processador), σ^S também executará. Não se determina previamente qual a tarefa de P que deverá migrar. Esta decisão é tomada em tempo de execução e depende das tarefas que estão ocupando cada processador no momento da tomada desta decisão.

Quando considera-se tarefas esporádicas, algumas destas podem não estar ativas durante alguns períodos de tempo. Neste caso, os subconjuntos produzidos pelo particionamento são definidos de forma que EDF local pode ser utilizado no lugar dos servidores QPS. Os detalhes sobre como é realizada esta adaptação são deixados para depois da definição precisa do conceito de quasi-partição.

5.3 O CONCEITO DE QUASI-PARTIÇÃO

Como mencionado anteriormente, QPS particiona o conjunto de tarefas a ser escalonado utilizando uma forma particular de particionamento denominada por **quasi-partição**, que definimos formalmente a seguir.

Definição 5.3.1 (Quasi-partição)**.** *Seja Γ um conjunto de tarefas ou de servidores a serem escalonados em m processadores idênticos. Uma quasi-partição de Γ com máximo de m partes, denotada por $\mathcal{Q}(\Gamma, m)$, é uma partição de Γ em que:*

(i) $|\mathcal{Q}(\Gamma, m)| \leq m$*;*

(ii) $\forall P \in \mathcal{Q}(\Gamma, m), 0 < \mathrm{R}(P) < 2$*; e*

(iii) $\forall P \in \mathcal{Q}(\Gamma, m), \forall \sigma \in P, \mathrm{R}(P) > 1 \Rightarrow \mathrm{R}(\sigma) > \mathrm{R}(P) - 1$

Cada elemento de P em $\mathcal{Q}(\Gamma, m)$ é um minor execution set (no caso de $\mathrm{R}(P) \leq 1$) ou um major execution set (quando $\mathrm{R}(P) > 1$).

A condição *(i)* desta definição restringe a quantidade de *execution sets* em uma quasi-partição, desconsiderando particionamentos cuja cardinalidade seja superior a quantidade de processadores necessários ao escalonamento de Γ. A condição *(ii)* garante que cada *execution set* em uma quasi-partição não requer mais do que dois processadores para ser escalonado corretamente. Quando mais do que um processador for necessário para o escalonamento de algum P em $\mathcal{Q}(\Gamma, m)$, a condição *(iii)* estabelece que o recurso extra necessário deve ser inferior à demanda de qualquer dos servidores em P. É importante ressaltar que esta última propriedade é a pedra angular que possibilita QPS lidar com tarefas esporádicas, como será detalhado na seção Section 5.4.2.

Existem diversas possíveis formas de se quasi-particionar um conjunto de servidores Γ. A seguir, são apresentadas duas formas de quasi-particionamento.

- **FFD**. Este método de quasi-particionamento inicia aplicando-se o algoritmo de bin packing First-Fit com as tarefas em ordem decrescente de utilização. Isto é, as tarefas são empacotadas em *execution sets* de forma que inicialmente nenhum *execution set* contenha um conjunto de tarefas com uma utilização total que exceda ao que pode ser tratado por um único processador. FFD empacota cada tarefa, uma de cada vez e em ordem decrescente de utilização, dentro do primeiro *execution set* em que ela caiba, ou abre um novo *execution set* se a tarefa não couber em nenhum dos *execution sets* já abertos, sendo a quantidade de *execution sets* limitada a m. Se uma tarefa não couber em nenhum dos m *execution sets*, ela será empacotada no *execution set* que possuir o *maior* espaço remanescente de forma a minimizar excedente à capacidade de um processador.

  A aplicação do algoritmo FFD é ilustrada pela Figura 5.3 em que oito tarefas são distribuídas por três *execution sets*. Nesta ilustração, a maior tarefa (com utilização igual a 0.80) é empacotada no primeiro *execution set*. Se a segunda maior tarefa (com utilização igual a 0.60) também fosse empacotada no primeiro *execution set*, a utilização deste iria ultrapassar 100%, portanto esta tarefa é empacotada no segundo *execution set*. A terceira maior tarefa (com utilização 0.40) também não cabe inteiramente no primeiro *execution set*, mas o segundo *execution set* pode recebê-la sem que o total das suas tarefas ultrapasse 100%. A quarta tarefa (com utilização 0.35) não cabe nos dois primeiros *execution sets*, sendo empacotada no terceiro *execution set*. O mesmo ocorre com a quinta e com a sexta tarefa que possuem respectivamente utilizações 0.30 e 0.25. A sétima tarefa, com utilização 0.15, consegue ser empacotada logo no primeiro *execution set*. A última tarefa, também com utilização 0.15, não pode ser empacotada em nenhum dos três *execution sets* sem que a utilização total destes ultrapasse a 100%. Assim, o *execution set* com mais espaço remanescente (o terceiro) receberá esta última tarefa.

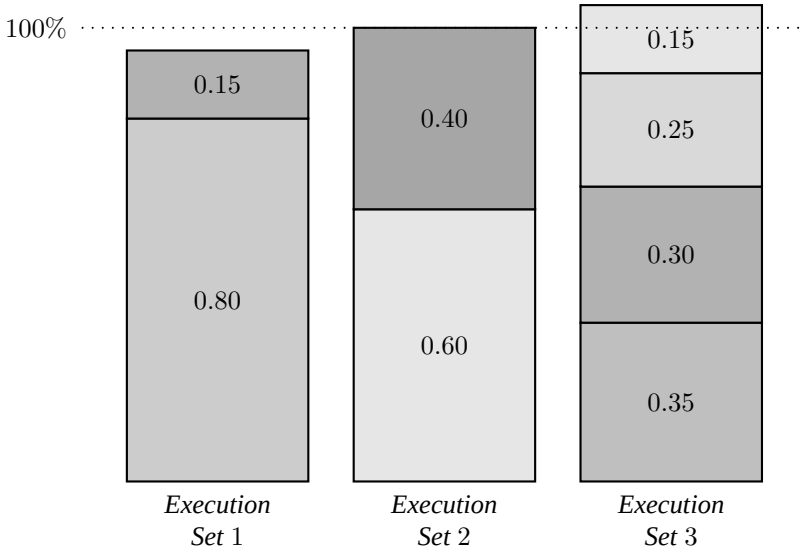


Figura 5.3. Ilustração do algoritmo de quasi-particionamento FFD aplicado em um conjunto de tarefas com utilizações 0.80, 0.60, 0.40, 0.35, 0.30, 0.25, 0.15, 0.15 em três *execution sets*.

- **EFD**. Com a intenção de construir *execution sets* com composições semelhantes, foi criado o algoritmo *Evenly-Fit Decreasing*. Da mesma forma que o FFD, as tarefas são organizadas em ordem decrescente de utilização. Os m *execution sets* formam uma lista circular em que considera-se o primeiro *execution set* como sucessor imediato do último *execution set*. Todos os m *execution sets* são abertos e o primeiro entre eles é selecionado. Cada tarefa é colocada no *execution set* atual se após esta inclusão ele permanecer necessitando de um único processador para ser corretamente escalonado. Caso isto não ocorra, toma-se o próximo *execution set* e repete-se o teste para a tarefa. Se todos os *execution sets* forem testados para alguma tarefa e ela não puder ser inserida em nenhum deles, esta tarefa é inserida no *execution set* seguinte ao da última alocação bem sucedida e o algoritmo continua com o teste da próxima tarefa no próximo *execution set*.

  A Figura 5.4 ilustra a aplicação deste algoritmo com o mesmo conjunto de tarefas utilizado na Figura 5.3. A tarefa com maior utilização (0.80) é empacotada no primeiro *execution set*, a segunda maior tarefa (com utilização 0.60) é empacotada no segundo *execution set* e a terceira maior tarefa (com utilização 0.40) é empacotada no terceiro *execution set*. Como o terceiro *execution set* é o último, o próximo a receber uma tarefa deveria ser novamente o primeiro *execution set*. No entanto, a quarta maior tarefa (com utilização 0.35) não cabe inteiramente neste *execution set* sendo empacotada no *execution set* seguinte. Seguindo a sequencia de *execution sets*, a quinta tarefa (com utilização 0.30) é empacotada no terceiro *execution set* fazendo com que o próximo a receber uma tarefa seja o primeiro *execution set*. No entanto, a sexta tarefa possui utilização 0.25 e não pode ser empacotada nem no primeiro nem no segundo *execution set* sem que estes ultrapassem uma utilização total de 100%. Assim, a sexta tarefa é empacotada no terceiro *execution set*. O próximo *execution set* a receber uma tarefa é o primeiro, e ele recebe a

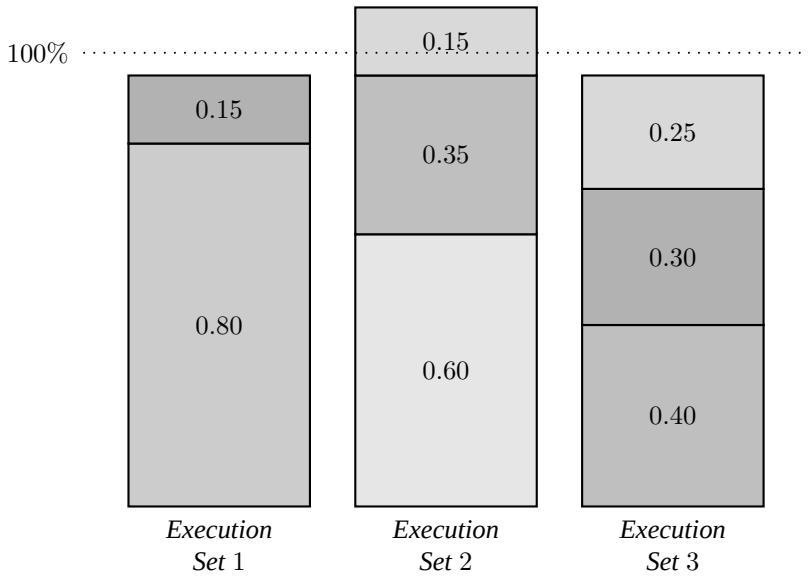


Figura 5.4. Ilustração do algoritmo de quasi-particionamento EFD aplicado em um conjunto de tarefas com utilizações 0.80, 0.60, 0.40, 0.35, 0.30, 0.25, 0.15, 0.15 em três *execution sets*.

sétima tarefa que possui utilização 0.15. A última tarefa, com utilização 0.15 não cabe inteiramente em nenhum dos três *execution sets*, sendo empacotada no segundo *execution set* por ser este o *execution set* seguinte ao último que havia recebido uma tarefa.

Pode-se observar que, em ambas as heurísticas, se cada *execution set* puder ser escalonado em um único processador, será obtido um particionamento adequado para as n tarefas em m processadores. Além disso, não é possível que todos os *execution sets* necessitem de mais do que um processador pois isso levaria a um particionamento de Γ em m subconjuntos, cada qual com utilização superior a um. Teria-se $\mathrm{R}(\Gamma) > m$, e o sistema seria infactível. Além disso, pode-se notar que as condições *(i)-(iii)* da Definição 5.3.1 são satisfeitas. Em particular, *(iii)* é garantido pois a tarefa que possui a menor utilização em um *execution set* é a última tarefa a ser incluída neste *execution set*, já que as tarefas são tratadas em ordem decrescente, e o tamanho desta tarefa é superior ao excesso que passa a haver neste respectivo *execution set*, *i.e.*, $\mathrm{R}(\sigma) > \mathrm{R}(P) - 1$. Logo, considerando as tarefas previamente ordenadas, os algoritmos descritos levam a uma quasi-partição correta e custam $O(nm)$ passos.

5.4 ESCALONAMENTO QUASI-PARTICIONADO

O algoritmo QPS possui três componentes básicos: o *Execution Set Alocator*, o *Manager*, e o *Dispatcher*. O *Execution Set Alocator* é o responsável pelo quasi-particionamento off-line das tarefas em *execution sets* e pela alocação destes aos processadores. O *Manager* ativa e desativa os servidores QPS (Definição 5.2.1) em resposta às alterações na demanda do sistema em tempo de execução. O *Dispatcher* é o responsável pelo escalonamento dos servidores que estão ativos no sistema.

Algorithm 5.1: Execution Set Alocator

Entrada: Conjunto Γ composto por n servidores que deverão ser escalonados em m processadores idênticos

Saída: Associação entre processadores e *execution sets*

```
1  j ← 1
2  P ← Q(Γ, m)
3  while ∃P ∈ P, R(P) > 1 do
4      Γ ← ∅
5      foreach P ∈ P tal que R(P) > 1 do
6          x ← R(P) − 1
7          Γ ← Γ ∪ {σ_{n+j}:(x, P)}
8          Associe o execution set P ao processador dedicado j
9          j ← j + 1
10     foreach P ∈ P de forma que R(P) ≤ 1 do
11         Γ ← Γ ∪ P
12     P ← Q(Γ − j + 1)
13 foreach P ∈ P do
14     Associe o execution set P ao processador dedicado j
15     j ← j + 1
```

5.4.1 O Execution Set Alocator

O *Execution Set Alocator* é um procedimento *off-line* que associa *execution sets* com processadores, a partir do quasi-particionamento obtido por uma sub-rotina $\mathcal{Q}(\cdot)$ (Definição 5.3.1), conforme especificado pelo Algoritmo 5.1. Começando com uma quasi-partição de um conjunto de n tarefas (linha 2), o *Execution Set Alocator* aloca um processador inteiro para qualquer *major execution set* P e define um servidor externo σ_{n+j} responsável pela reserva de tempo de processamento para a execução de P em um outro processador (linhas 6-7). Note que este é um procedimento iterativo. Os servidores externos dos *major execution sets* são unidos ao conjunto Γ (linha 7) composto por tarefas/servidores em *minor execution sets* (linhas 10-11). Este conjunto é novamente quasi-particionado (linha 12), e o procedimento é iterado. Uma vez que não reste mais nenhum *major execution set*, cada *minor execution set* receberá o seu próprio processador (linhas 13-15). Note que quando a execução inicial de $\mathcal{Q}(\Gamma, m)$ na linha 2 retorna apenas *minor execution sets*, QPS é reduzido a EDF particionado.

É importante destacar que se P for um *major execution set*, ele será explicitamente associado a um processador que executa exclusivamente tarefas em P (linha 8), denominado *processador dedicado* a P. A capacidade adicional de processamento necessária à execução de P, $\mathrm{R}(P) - 1$, é reservada pelo servidor externo em um outro processador. Este servidor, por sua vez, pode vir a ser incluído em um outro *major execution set* e, como resultado, P pode terminar sendo executando em dois ou mais processadores, denominados *processadores compartilhados* de P. Apesar de a alocação de *processadores compartilhados* não aparecer

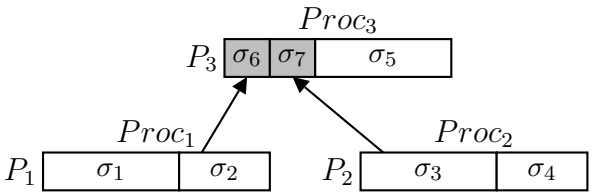


Figura 5.5. Ilustração da hierarquia de processadores definida pelo Algoritmo 5.1 para o Exemplo 5.4.1. Servidores σ_6 e σ_7 são externos e definem reservas para a alocação dos servidores *master* dos *major execution sets* P_1 e P_2.

explicitamente no Algoritmo 5.1, ela é realizada pelas linhas 8 e 14, sempre que o *execution set* considerado contiver um servidor externo criado pela linha 7.

Exemplo 5.4.1. *Seja* $\Gamma = \{\sigma_1, \sigma_2, \ldots, \sigma_5\}$ *um conjunto de servidores com* $\mathrm{R}(\sigma_i) = 0.6$ *para* $i = 1, 2, \ldots, 5$ *escalonados em três processadores.*

O comportamento do *Execution Set Alocator* é ilustrado pelo Exemplo 5.4.1. Seja $\mathcal{P} = \{\{\sigma_1, \sigma_2\}, \{\sigma_3, \sigma_4\}, \{\sigma_5\}\}$ a quasi-partição inicial definida pela linha 2. Como existem dois *major execution sets* em $\mathcal{P}$, o laço existente nas linhas 5-9 é executado duas vezes, os dois primeiros processadores são dedicados aos *major execution sets*, digamos $P_1 = \{\sigma_1, \sigma_2\}$ e $P_2 = \{\sigma_3, \sigma_4\}$, e os servidores externos $\sigma_6 = (0.2, \{\sigma_1, \sigma_2\})$ e $\sigma_7 = (0.2, \{\sigma_3, \sigma_4\})$ são definidos na linha 7. Ao final da segunda iteração do laço `while`, $\mathcal{P} = \{\sigma_5, \sigma_6, \sigma_7\}$, o qual é associado ao terceiro processador. No final do procedimento, os processadores 1 e 2 estão respectivamente dedicados aos *major execution sets* P_1 and P_2, e o terceiro processador é dedicado ao *minor execution set* $P_3 = \{\sigma_5, \sigma_6, \sigma_7\}$. O processador 3 também pode ser considerado como sendo o processador compartilhado por P_1 e P_2.

Note que o *Execution Set Alocator* define uma hierarquia entre os processadores. No Exemplo 5.4.1, os dois processadores (dedicados) estão ligados ao terceiro processador (compartilhado) através dos servidores externos (Figura 5.5).

O *Execution Set Alocator* executa em tempo polinomial. O tamanho de $\mathcal{P}$ no pior caso é $|\mathcal{P}| = \lceil \mathrm{R}(\Gamma) \rceil \leq m$, que é correspondente a maior quasi-partição possível pela Definição 5.3.1. Ao serem definidos os servidores, a utilização de $\mathcal{P}$ decresce em pelo menos 1 (lines 6 e 7) a cada iteração do laço `while`. Portanto o laço `while` custa $O(m)$ passos. Como uma rotina de quasi-particionamento pode ser implementada em $O(nlogn + nm)$, o laço `while` executa em $O(nmlogn + nm^2)$. Além disso, as linhas 13-15 custam $|\mathcal{P}|$ passos. Portanto, levando em consideração o quasi-particionamento realizado na linha 2, o procedimento completo executa em $O(nmlogn + nm^2) = O(mn^2)$. Como $m = O(n)$, o tempo de execução do algoritmo é limitado por $O(n^3)$.

Pode-se observar a seguinte propriedade:

Lema 5.4.1. *O Algoritmo 5.1 será bem sucedido em associar qualquer conjunto de servidores* Γ *com* $m \geq \lceil \mathrm{R}(\Gamma) \rceil$ *processadores.*

Demonstração. Pela definição de quasi-partição (Definição 5.3.1), sabe-se que o número de *execution sets* gerados pelo quasi-particionamento inicial realizado na linha 2 é de, no máximo,

m. Sabe-se, ainda, que o número de *execution sets* gerados pelos quasi-particionamentos da linha 12 decrescem em pelo menos uma unidade a cada iteração, já que j é incrementado em uma unidade a cada *major execution set* processado. Uma vez que cada *major execution set* recebe o seu próprio processador, o laço `while` pode alocar o máximo de m processadores. Suponha que $k \leq m$ processadores são alocados durante todas as iterações do laço `while`. Isso significa que, após a última execução da linha 12, $\lceil \mathrm{R}(\mathcal{P}) \rceil$ será no máximo igual a $k' \leq m - k$. Pela Definição 5.3.1, sabe-se que não pode haver mais do que k' *execution sets* em $\mathcal{P}$ e as linhas 13-15 alocam um processador para cada um deles. □

5.4.2 O Manager

Seja P um *major execution set* associado a um processador dedicado e com um servidor externo responsável por sua execução em um processador compartilhado. De acordo com o modelo adotado com tarefas esporádicas, é possível que os servidores em P possam executar exclusivamente no processador dedicado a P durante um certo intervalo de tempo, sem que o processador compartilhado seja necessário. Neste caso, o *Manager* desativa os servidores QPS (Definição 5.2.1) encarregados da execução de P de forma que P seja simplesmente escalonado por EDF durante este intervalo, semelhante ao que é feito com um *minor execution set*.

Por outro lado, se P está sendo escalonado por EDF, a chegada de um job esporádico pode tornar necessário o uso de parte (ou de todo) o tempo de execução reservado para o servidor externo no seu processador compartilhado. Nesse caso, o *Manager* ativa o conjunto de servidores QPS para P, utilizando a reserva do servidor externo para a definição do servidor *master* responsável por P.

Portanto, um *major execution set* P pode ser escalonado em dois diferentes modos, EDF e QPS. No modo QPS, os servidores QPS escalonam os servidores em P em dois ou mais processadores, enquanto no modo EDF os servidores em P são simplesmente escalonados por EDF em um único processador. A transição entre estes dois modos é chamada de *mudança de modo*.

Note que uma *mudança de modo* realizada por um *major execution set* pode desencadear uma *mudança de modo* em outros *major execution sets*. Isso acontece na ativação e na desativação de um servidor *master* que faz parte de outro *major execution set*.

O Algoritmo 5.2 descreve o procedimento executado pelo *Manager*. A cada instante t em que ocorre a liberação ou o *deadline* de um job de servidores contidos em um *major execution set* P definido pelo Algoritmo 5.1, o Algoritmo 5.2 (linha 2) determina qual modo deverá ser utilizado através da chamada da função *qps_mode(P, t)*. Essa função retorna `verdadeiro` se os servidores QPS devem ser ativados e retorna `falso` no caso contrário. Quando os servidores QPS forem ativados, P é bi-particionado em P^A e P^B, escolhendo como único componente para P^A um entre os servidores com jobs liberados no instante t (linha 5). Em seguida, a função *qps_rate(P,t)* é chamada na linha 8 para definir qual será a taxa utilizada na ativação

Algorithm 5.2: Manager

Entrada: *Major execution set* P; instante em que ocorre uma liberação ou um *deadline* de algum servidor em P
Saída: Ativação/desativação dos servidores QPS associados com P

```
1  Seja t um instante em que ocorre uma liberação ou um deadline
2  if qps_mode(P, t) (// modo QPS) then
3      if servidores QPS estão inativos no instante t then
4          Seja σ ∈ P o servidor liberado no instante t
5          P^A ← {σ}
6          P^B ← P\{σ}
7          Ative os servidores QPS
8      x ← qps_rate(P, t) − 1
9      σ^M:(x, P)
10     σ^S:(x, P)
11     σ^A:(R(P^A) − x, P^A)
12     σ^B:(1 − R(P^A), P^B)
13 else // modo EDF
14     Desative os servidores QPS associados com P
```

dos servidores QPS em t, sendo que a taxa do servidor *master* não deve ser superior à taxa do servidor externo definido pelo Algoritmo 5.1. Como a função *qps_rate(P,t)* não retorna valores superiores a $\mathrm{R}(P)$, os servidores QPS correspondentes estarão adequados à hierarquia de processadores definida pelo Algoritmo 5.1.

Existem algumas opções para realizar a troca entre os modos QPS e EDF. Estas opções serão apresentadas a seguir.

5.4.2.1 Estratégias de Adaptação

Foram definidas três estratégias de adaptação diferentes: conservadora com a taxa total (**CF** - *Conservative Full-Rate*); baseada em taxa com a taxa total (**RF** - *Rate-based with Full-rate*) e baseada em taxa com taxa parcial (**RP** - *Rate-based with Partial rate*).

A estratégia CF é conservadora e assume que sempre que todos os servidores de um *major execution set* P estão ativos, então é preciso que os servidores QPS associados a P sejam ativados provendo uma taxa conjunta igual a ($\mathrm{R}(P)$). Entretanto, se houver algum servidor inativo em P, então este será escalonado por EDF local em seu processador dedicado. Nas estratégias RF e RP, a taxa exigida pelos servidores ativos em P no instante t em que ocorrer a liberação ou o *deadline* de um job, denominada $\rho(P, t)$, é calculada de forma exata (a Seção 5.4.2.2 oferecerá detalhes sobre como essa taxa é calculada). Se for necessário mais do que um processador para o correto escalonamento de P, isto é $\rho(P, t) > 1$, os servidores QPS serão ativados pelo *Manager*. Caso contrário, os servidores em P serão escalonados por EDF local em seu processador dedicado. A diferença entre RF e RP é a taxa utilizada para configurar

os servidores QPS quando estes forem necessários. Na estratégia RF os servidores QPS são sempre recarregados com a sua taxa total, enquanto na estratégia RP os servidores QPS podem ser recarregados de forma parcial.

A seguir são especificadas as funções `qps_mode(P,t)` e `qps_rate(P,t)` de acordo com a estratégia de adaptação escolhida. A primeira determina qual modo (QPS ou EDF) deve ser utilizado. Note que esta função indica que o modo QPS deve ser utilizado quando todos os servidores estão ativos para CF e, alternativamente, quando $\rho(P,t) > 1$ para RF ou RP.

Function qps_mode(P, t)

$A \leftarrow$ todos os servidores de$P$que estão ativos no instante t
case *CF:* **return** $A = P$

case *RF ou RP:* **return** $A = P \wedge \rho(P,t) > 1$

A função `qps_rate(P,t)` só será chamada quando o modo atual for o modo QPS (linha 8 do Algoritmo 5.2). Quando o modo atual for EDF, P estará sendo escalonado por EDF e não existirá a necessidade de se calcular uma taxa de execução.

Function qps_rate(P, t)

`// qps_mode(P,t) igual a Verdadeiro`
case *CF or RF:* **return** $R(P)$

case *RP:* **return** $\rho(P,t)$

5.4.2.2 Cálculo de $\rho(P,t)$

Para calcular $\rho(P,t)$, o *Manager* estima o máximo de *tempo de serviço* disponível a partir do instante t, assumindo que todos os servidores em P estarão ativos e serão executados a partir de $D(P,t)$ a uma taxa conjunta de $\mathrm{R}(P)$. Fazendo isso, o *Manager* determina qual a demanda dos servidores em P existente no instante t e que pode ser postergada para depois de $D(P,t)$. Subtraindo-se esta quantidade da demanda em t, o que resta é a parte urgente que deve ser executada durante o intervalo $[t, D(P,t))$. Finalmente, $\rho(P,t)$ é obtida dividindo-se essa demanda urgente por $D(P,t) - t$. Uma vez que cada servidor em P pode ter sua própria demanda urgente, $\rho(P,t)$ é o máximo dessas taxas obtidas para cada servidor P. Este procedimento é detalhado a seguir.

Denote o conjunto de servidores que, no instante t, possuem prioridade igual ou superior ao servidor $\sigma_i \in P$ como

$$\mathrm{hp}_i(P,t) = \{\sigma_j \in P, D(\sigma_j,t) \leq D(\sigma_i,t)\}$$

Para todo servidor $\sigma_i \in P$ que esteja ativo no instante t, o sistema deve executar todos os jobs em $\text{hp}_i(P,t)$ durante $[t, D(\sigma_i,t))$. Isso representa o restante da carga exigida pelos jobs já liberados em t mas ainda não terminados, somado aos jobs a serem liberados depois de t e que possam interferir na execução do job de σ_i. Para todo servidor $\sigma_i \in P$, esta demanda é denotada por $\phi_i(P,t)$:

$$\phi_i(P,t) = \sum_{\sigma_j \in \text{hp}_i(P,t)} \text{E}(\sigma_j,t) + \text{R}(\sigma_j)(D(\sigma_i,t) - D(\sigma_j,t))$$

O *tempo de serviço* disponível entre os instantes $D(P,t)$ e $D(\sigma_i,t)$ em relação a qualquer servidor $\sigma_i \in P$ é denotado por $\psi_i(P,t)$. Assumindo que os servidores QPS fornecem a sua taxa máxima de execução $\text{R}(P)$ durante o intervalo $[D(P,t), D(\sigma_i,t))$ e que os servidores em P executam a esta mesma taxa, obtemos

$$\psi_i(P,t) = \text{R}(P)(D(\sigma_i,t) - D(P,t))$$

A diferença $\phi_i(P,t) - \psi_i(P, D(P,t))$ fornece a demanda urgente que o servidor σ_i deve executar no intervalo $[t, D(P,t))$. Portanto, a menor taxa de execução necessária para os servidores em P durante $[t, D(P,T))$ é

$$\rho(P,t) = \frac{\max_{\sigma_i \in P}\{\phi_i(P,t) - \psi_i(P,t)\}}{D(P,t) - t} \tag{5.1}$$

5.4.3 O Dispatcher

O *Dispatcher*, descrito pelo Algoritmo 5.3, visita cada processador j para o qual exista ao menos um servidor ativo (linhas 3-4); seleciona um servidor para cada processador (linhas 5-8); seleciona a tarefa que deve ser executada (linhas 9-14); e, finalmente, executa o conjunto de tarefas selecionadas (linha 16). Este procedimento é executado sempre que uma tarefa (ou servidor) libera um novo job e sempre que um job é concluído.

Na Seção 5.4.1 foi visto que o Algoritmo 5.1 define uma hierarquia de processadores. O Algoritmo 5.3 faz uso dessa hierarquia, visitando os processadores na ordem inversa a que foi produzida pelo Algoritmo 5.1. Como resultado, os servidores *masters* são selecionados antes dos respectivos *slaves*, garantindo a sua execução simultânea.

Considerando um *execution set* específico P alocado ao processador j, as regras de execução são as seguintes. Independentemente de P ser um *major execution set* ou um *minor execution set*, se P estiver no modo EDF então os servidores em P serão escalonados por EDF no processador j, de forma semelhante a um sistema com único processador. Por outro lado, se P for um *major execution set* executando no modo QPS, seus servidores QPS serão escalonados de acordo com a hierarquia existente entre cada *master* e seu *slave*, ou seja, sempre

Algorithm 5.3: Dispatcher

```
1  Seja t o instante atual e d o próximo deadline após t
2  Seja k o último processador alocado pelo Algoritmo 5.1
3  for j ← k, k − 1, . . . , 1 do
4      if no instante t existe ao menos um servidor ativo no processador j then
5          if existe um servidor slave no processador j cujo respectivo master já foi
           previamente selecionado then
6              Selecione o servidor slave σ no processador j
7          else
8              Selecione um servidor não slave σ no processador j em ordem arbitrária
9          while σ é uma tarefa do
10             P ← cli(σ)
11             if σ não é um servidor slave then
12                 Selecione σ′ em P utilizando EDF
13             else
14                 Selecione σ′ em P^A se o cliente selecionado pelo seu master estiver em
                   P^B ou vice-versa
15             σ ← σ′
16 Execute todas as tarefas nos processadores em que elas foram selecionadas
```

que o servidor *master* executar no processador externo de P, o seu servidor *slave* associado executará no processador dedicado j, de modo a assegurar a execução paralela exigida por P (linhas 5-6). Sempre que um servidor *master* não estiver sendo executado, o respectivo servidor *slave* também não executará, e os dois servidores QPS dedicados serão selecionados em ordem arbitrária, uma vez que eles compartilham os mesmos *deadlines* (linha 8). Em todos os casos apresentados os servidores selecionam seus clientes por EDF.

Uma vez que um servidor é selecionado em um determinado processador j, a tarefa que deverá ser executada em j deve ser escolhida. Como um servidor pode encapsular outros servidores, uma cadeia recursiva de servidores deve ser seguida até que uma tarefa seja atingida, isto é feito pelas linhas 9-15. Servidores *master* e *slave* podem servir a quaisquer clientes de um *execution set*, mas o mesmo cliente não pode ser executado simultaneamente em dois processadores diferentes. A linha 14 impede que isto aconteça.

5.4.4 Ilustração

O comportamento da política de escalonamento QPS é ilustrado pelo Exemplo 5.4.2, construído a partir de uma ligeira alteração no Exemplo 5.2.1 para que o sistema utilize completamente dois processadores e trate com tarefas esporádicas. A quasi-partição assumida para este exemplo é $\mathcal{Q}(\Gamma, 2) = \{P_1, P_2\}$ com $P_1 = \{\tau_1, \tau_2, \tau_3\}$ e $P_2 = \{\tau_4\}$. O Algoritmo 5.1 aloca P_1 em um processador, sendo que o seu servidor *master* compartilha o segundo processador com

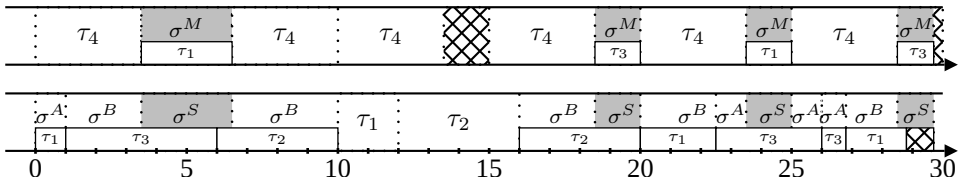


Figura 5.6. Escalonamento quasi-particionado para o Exemplo 5.4.2 onde $\mathcal{Q}(\Gamma, 2) = \{\{\tau_1, \tau_2, \tau_3\}, \{\tau_4\}\}$, com a estratégia de adaptação CF. O segundo job de τ_3 chega atrasado em $t = 16$.

τ_4. É assumido que todas as quatro tarefas liberam seus primeiros jobs no instante 0. O segundo job de τ_3 chega no instante $t = 16$, enquanto as demais tarefas chegam periodicamente. As estratégias de adaptação para este exemplo são consideradas separadamente.

Exemplo 5.4.2. *Considere o conjunto de tarefas esporádicas* $\Gamma = \{\tau_1 : (6, 15)$,$\tau_2 : (12, 30)$, τ_3:$(5, 10)$,τ_4:$(3.5, 5)\}$ *a ser escalonado em dois processadores.*

Estratégia de Adaptação CF. No instante 0, o Algoritmo 5.2 define a ativação dos servidores QPS encarregados da execução de P_1, ou seja σ^M, σ^S, σ^A e σ^B, uma vez que todas as tarefas em P_1 estão ativas e, portanto, `qps_mode`$(P_1, 0) =$ `verdadeiro`. Como as tarefas em P_1 foram ativadas simultaneamente, a distribuição destas por σ^A e σ^B é arbitrária, sendo exigido apenas que uma única tarefa seja atribuída à σ^A. Nesta ilustração, tem-se que inicialmente σ^A atenderá $\{\tau_1\}$ e σ^B atenderá $\{\tau_2, \tau_3\}$. A carga total destes servidores QPS é definida pelo Algoritmo 5.2 como sendo igual a `qps_rate`$(P_1, 0) = \mathrm{R}(P_1) = 1.3$, criando os servidores σ^M:$(0.3, P_1)$, σ^S:$(0.3, P_1)$, σ^A:$(0.1, \{\tau_1\})$ e σ^B:$(0.6, \{\tau_2, \tau_3\})$. Assim, como o primeiro *deadline* das tarefas em P_1 ocorre no instante $t = 10$, temos que os servidores σ^M, σ^S, σ^A e σ^B receberão respectivamente cargas iniciais iguais a 3, 3, 1 e 6. A Figura 5.6 apresenta o escalonamento produzido por QPS para este exemplo.

De acordo com o Algoritmo 5.3, como as decisões de escalonamento devem ser tomadas na ordem decrescente do índice dos processadores, as decisões de escalonamento devem ser iniciadas pelo segundo processador. No instante $t = 0$ a tarefa τ_4 possui a maior prioridade no segundo processador (considerando EDF) por isso ela é escolhida para executar durante o intervalo $[0, 3.5)$. Para o primeiro processador, tanto σ^A quanto σ^B poderiam ser selecionados para executar em $t = 0$, já que possuem *deadlines* iguais. A Figura 5.6 mostra o caso em que σ^A é selecionado e executa o seu único cliente, τ_1, até o instante $t = 1$. Após $t = 1$, o servidor σ^B inicia a execução dos seus clientes. A tarefa τ_3 possui menor *deadline* se comparada com τ_2 e, portanto, é escolhida por σ^B para executar até o instante $t = 3.5$, quando o servidor σ^S começa a executar induzido pelo escalonamento do servidor σ^M no segundo processador. Durante o intervalo $[3.5, 6.5)$, σ^M e σ^S executam um cliente cada, selecionados das partições $\{\tau_1, \tau_2\}$ e $\{\tau_3\}$, respectivamente. Como τ_1 possui prioridade maior do que τ_2 (por EDF) e ele já se encontra executando no processador dedicado, ele permanece executando no mesmo processador, só que agora servido por σ^S. Note que esta substituição de servidores não provoca uma preempção na tarefa servida. A tarefa τ_3 migra para ser servida pelo servidor σ^M no

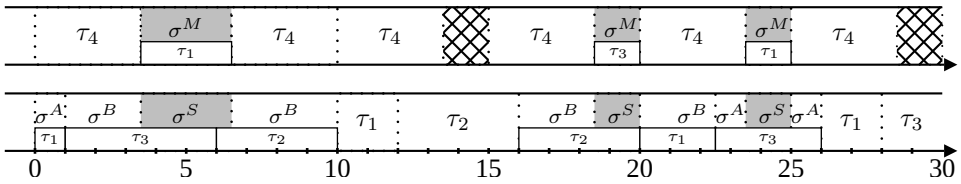


Figura 5.7. Escalonamento quasi-particionado para o Exemplo 5.4.2 onde $\mathcal{Q}(\Gamma, 2) = \{\{\tau_1, \tau_2, \tau_3\}, \{\tau_4\}\}$, com a estratégia de adaptação RF. O segundo job de τ_3 chega atrasado em $t = 16$.

segundo processador. O escalonamento apresentado na figura prossegue de forma semelhante até $t = 10$, quando a tarefa τ_3 atrasa e torna-se inativa. Até a chegada deste job atrasado (que ocorre em $t = 16$), os servidores QPS são mantidos inativos pelo Algoritmo 5.2 pois tem-se que $\texttt{qps\_mode}(P_1, t) = \texttt{falso}$, para $t \in [10, 16)$. Com isso, durante o intervalo $[10, 16)$ as tarefas τ_1 e τ_2 são escalonadas por EDF no primeiro processador.

No instante $t = 16$, quando o segundo job de τ_3 chega, todas as tarefas em P_1 estão novamente ativas e, portanto, $\texttt{qps\_mode}(P_1, 16) = \texttt{verdadeiro}$. De acordo com o Algoritmo 5.2, os servidores QPS são ativados com $\sigma^M : (0.3, P_1)$, $\sigma^S : (0.3, P_1)$, $\sigma^A : (0.2, \{\tau_3\})$ e $\sigma^B : (0.5, \{\tau_1, \tau_2\})$. Observe que σ^A serve apenas à τ_3 que foi a última tarefa a ficar ativa. As tarefas τ_1 e τ_2 são os novos clientes de σ^B, como descrito pelo Algoritmo 5.2. Neste instante, os servidores QPS liberam seus jobs com o mesmo *deadline* $D(\sigma, 16) = 26$ e com a sua taxa máxima, ou seja $\sigma^M, \sigma^S, \sigma^A$ e σ^B liberam seus jobs com tempos de execução $3, 3, 2$, e 5 respectivamente. As próximas decisões de escalonamento seguem procedimento similar.

A estratégia de adaptação adotada neste exemplo é conservadora, estabelecendo reservas nos servidores mesmo quando estas não serão necessariamente utilizadas. Isso pode ser constatado pelo intervalo $(28.8, 30]$ em que existe um servidor σ^S sendo escalonados sem tarefas clientes para serem atendidas.

Estratégia de Adaptação RF. Apesar de aplicar instrumentos diferentes da estratégia CF, a estratégia de adaptação RF produz escalonamento igual dentro de um período inicial, como pode ser visto na Figura 5.7. O Algoritmo 5.2 igualmente define a ativação dos servidores QPS no instante $t = 0$ por obter $\texttt{qps\_mode}(P_1, 16) = \texttt{verdadeiro}$. Entretanto, isto ocorre porque, além de todas as tarefas em P_1 estarem ativas, tem-se também que $\rho(P_1, 0) = 1.3 > 1$. Todo o escalonamento prossegue como na estratégia CF e mais à frente, em $t = 10$, os servidores QPS tornam-se igualmente inativos devido ao atraso de τ_3.

No instante $t = 16$, quando o segundo job de τ_3 chega, todas as tarefas em P_1 estão ativas novamente e $\rho(P_1, 16) = 1.18 > 1$, portanto tem-se $\texttt{qps\_mode}(P_1, 16) = \texttt{verdadeiro}$ e os servidores QPS são novamente ativados com a sua carga total. De acordo com o Algoritmo 5.2, os servidores QPS são ativados com $\sigma^M : (0.3, P_1)$, $\sigma^S : (0.3, P_1)$, $\sigma^A : (0.2, \{\tau_3\})$ e $\sigma^B : (0.5, \{\tau_1, \tau_2\})$ e o escalonamento prossegue idêntico à estratégia CF.

Em $t = 26$ a diferença entre as estratégias CF e RF aparece. O cálculo $\rho(P_1, 16) = 1.18$ indicava que seria necessário executar P_1 ao menos a uma taxa de 1.18 para que ele pudesse, a partir de $D(P_1, 16) = 26$, ser executado a uma taxa de 1.3 e, ainda assim, atender aos seus

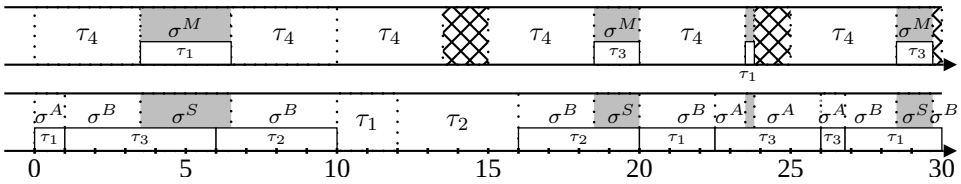


Figura 5.8. Escalonamento quasi-particionado para o Exemplo 5.4.2 onde $\mathcal{Q}(\Gamma, 2) = \{\{\tau_1, \tau_2, \tau_3\}, \{\tau_4\}\}$, com a estratégia de adaptação RP. O segundo job de τ_3 chega atrasado em $t = 16$.

deadlines. Como os servidores QPS são totalmente recarregados na estratégia de adaptação RF, P_1 pode executar no intervalo $(16, 26]$ a uma taxa superior, antecipando computação que poderia ter sido postergada para o intervalo $(26, 30]$. Com isso, quando em $t = 26$ o Algoritmo 5.2 recebe $\texttt{qps\_mode}(P_1, 16) = \texttt{falso}$ (pois $\rho(P_1, 26) = 1.00$) os servidores QPS ficam inativos e as tarefas em P_1 voltam a ser escalonadas por EDF.

Estratégia de Adaptação RP. Pode ser visto na Figura 5.8 que a estratégia RP segue idêntica à estratégia RF até o instante em que o job atrasado de τ_3 chega em $t = 16$. Neste momento, quando o Algoritmo 5.2 decide pela reativação dos servidores QPS, tem-se que $\texttt{qps\_rate}(P_1, 16) = 1.18$ e isso faz com que os servidores QPS sejam ativados como $\sigma^M : (0.18, P_1)$, $\sigma^S : (0.18, P_1)$, $\sigma^A : (0.32, \{\tau_3\})$ e $\sigma^B : (0.50, \{\tau_1, \tau_2\})$. Assim, $\sigma^M, \sigma^S, \sigma^A$ e σ^B liberam seus jobs com tempos de execução $1.8, 1.8, 3.2$ e 5.0, respectivamente, com *deadlines* $D(\sigma, 16) = 26$, ocupando apenas 18% do segundo processador no intervalo $(16, 26]$ e postergando a execução das tarefas o máximo possível sem o comprometimento da sua escalonabilidade. Em $t = 26$, todas as tarefas estão ativas e $\rho(P_1, 26) = 1.3$, fazendo com que os servidores QPS permaneçam ativos e executam na sua taxa máxima.

5.5 ANÁLISE DE ESCALONABILIDADE DO QPS

Como descrito na Seção 5.4, um *major execution set* P pode ser escalonado por um conjunto de servidores QPS ou por EDF. Portanto, é preciso comprovar que a combinação do uso de servidores QPS e do escalonamento por EDF produz um escalonamento válido. Para isso, inicialmente enunciamos o Lema 5.5.1 afirmando que o tempo de execução provido pelos servidores QPS para um *execution set* P será completamente consumido pelos clientes de P. Então, no Lema 5.5.2 é provada a correção das estratégias de adaptação. O Teorema 5.5.1 completa a prova, considerando todos os *execution sets* do sistema.

Para as provas a seguir, define-se a política de escalonamento QPS como sendo composta pela criação dos *execution sets* e sua alocação aos processadores através do Algoritmo 5.1, o gerenciamento dos servidores QPS pelo Algoritmo 5.2, e a liberação de jobs pelo Algoritmo 5.3. Enquanto um *major execution set* P estiver executando em modo QPS, os servidores em P estarão sendo executados por servidores QPS, e enquanto P estiver executando em modo EDF os seus servidores estarão sendo escalonados diretamente por EDF.

Lema 5.5.1. *Seja P um major execution set sob a política de escalonamento QPS. Considere*

$[t, D(P,t))$ *como um intervalo de tempo durante o qual* P *é escalonado no modo QPS e assuma que* $C \geq 0$ *é a carga de trabalho demandada pelos servidores em* P *no instante* t*. Se* r *for a taxa de execução provida pelos servidores QPS em* $[t, D(P,t))$*, com* $1 < r \leq \mathrm{R}(P)$*, então os servidores em* P *executam* $\min(r(D(P,t) - t), C)$ *durante* $[t, D(P,t))$.

Demonstração. Este lema é provado pelo cálculo do tempo em que é permitido que os servidores QPS executem seus clientes. Considere um intervalo de tempo $[t_0, t_1)$ de tal modo que $[t, D(P,t)) \subseteq [t_0, t_1)$ e todos os servidores QPS são ativados em t_0 e servem P continuamente em modo QPS durante $[t_0, t_1)$. No instante t_0, os servidores QPS são configurados como especificado pelas linhas 10-12 do Algoritmo 5.2. Seja σ o servidor liberado no instante t_0 e selecionado pelo *Manager* (linha 5) como único componente de P^A. Durante $[t_0, t_1)$, a bipartição escolhida no instante t_0 se manterá a mesma ao menos até t_1, e todas as tarefas clientes encapsuladas nos servidores em P permanecerão ativas, de outra forma ocorreria uma transição para o modo EDF durante $[t_0, t_1)$. Logo, durante $[t_0, t_1)$, qualquer *deadline* de um servidor em P também será um instante de liberação de jobs deste mesmo servidor. Considere σ^M, σ^S, σ^A, e σ^B estes servidores QPS ativados em t_0.

Agora, considere $x = r - 1$ o excesso para o *execution set* P, calculado no instante t pelo Algoritmo 5.2 em sua linha 8. A Definição 5.3.1 garante que $\mathrm{R}(\sigma) > \mathrm{R}(P) - 1$ e, portanto, $R(\sigma) > x$ já que, por hipótese, $r \leq \mathrm{R}(P)$. Sabe-se que durante o intervalo $[t_0, t)$ as execuções de σ^A e de σ^M forneceram exatamente $\mathrm{R}(\sigma)(t - t_0)$ aos jobs de σ. Além disso, $\mathrm{R}(\sigma)(D(\sigma,t) - t_0)$ é a demanda acumulada de σ desde a liberação do seu job em t_0 até o instante t. Assim,

$$\mathrm{R}(\sigma)(D(\sigma,t) - t_0) - \mathrm{R}(\sigma)(t - t_0) = \mathrm{R}(\sigma)(D(\sigma,t) - t) > x(D(P,t) - t)$$

Portanto, o tempo que falta para σ executar no instante t é maior do que o tempo de execução fornecido pelo servidor σ^M durante $[t, D(P,t))$. De acordo com o Algoritmo 5.1, sabe-se que σ^M está associado com P^A durante $[t_0, t_1)$. Assim, sempre que σ^M for selecionado para execução durante $[t, D(P,t))$, o job cliente liberado pelo servidor σ antes de t não terá sido completamente executado e pode ser executado com segurança, sem qualquer possibilidade de conflito com jobs de outros servidores clientes em P^B, pois os servidores em P^B poderão executar seguramente no processador dedicado ao *execution set* P. Portanto, ou toda a demanda C existente no instante t pode ser atendida pelos servidores QPS, ou seus jobs serão executados até sua conclusão, o que demora o tempo de $r(D(P,t) - t)$. □

Lema 5.5.2. *Seja* P *um major execution set sob a política de escalonamento QPS. Se todos os servidores QPS responsáveis por* P *atenderem aos deadlines, então todos os servidores em* P *também cumprirão os seus deadlines.*

Demonstração. Inicialmente este lema será provado para QPS utilizando a estratégia de adaptação RP, e depois o resultado será estendido para as demais estratégias de adaptação. Para comprovar a parte inicial do lema, será construído um cenário artificial em que é garantido que

o *Manager* sempre utilizará a Equação (5.1) quando calcular $\rho(P,t)$. Posteriormente esta restrição será removida. Em outras palavras, se não existirem servidores inativos em t, o cenário artificial consistirá na liberação de jobs em t com *deadline* em d, onde d é o próximo *deadline* após t, isto é $d = D(P,t)$. Procedendo desta forma, o valor obtido pela Equação (5.1) deverá aumentar (mas nunca diminuir) se comparada com a taxa real de execução necessária. Para evitar qualquer ambiguidade, será utilizado $\rho^*(P,t)$ como valor obtido pela Equação (5.1) no cenário artificial.

Como a Equação (5.1) recebe o máximo valor possível para os seus parâmetros de entrada, a taxa calculada no instante t satisfaz

$$\rho^*(P,t) \geq \frac{\phi_i(P,t) - \psi_i(P,t)}{d-t}$$

para qualquer servidor σ_i para o qual $D(\sigma_i,t) = d$. Além disso, para estes servidores $\psi_i(P,t) = 0$ e portanto

$$\phi_i(P,t) \leq \rho^*(P,t)(d-t)$$

Como σ_i possui o primeiro *deadline* após t e os servidores QPS fornecem uma taxa de execução igual a $\rho^*(P,t)$ no intervalo $[t,d)$, sabe-se, pelo Lema 5.5.1, que se $\rho^*(P,t) \leq \mathrm{R}(P)$, σ_i atenderá seu *deadline*. Portanto, σ_i somente não será capaz de cumprir seu *deadline* em d se $\rho^*(P,t) > \mathrm{R}(P)$.

Agora será mostrado que isso não ocorre, ou seja, para qualquer *deadline* d, $\rho^*(P,d) \leq \mathrm{R}(P)$. Suponha, por contradição, que d é o primeiro momento em que $\rho^*(P,d) > \mathrm{R}(P)$. Note que t é o último instante antes de d em que $\rho^*(P,t)$ foi calculado. Isto ocorre pois está garantido que todos os servidores estão ativos em d, mesmo que isto ocorra pela liberação de jobs artificiais. Pela definição de t e d, tem-se que $\rho^*(P,t) \leq \mathrm{R}(P)$. Além disso, pela Equação (5.1) e pela hipótese de contradição temos que

$$\phi_k(P,d) - \psi_k(P,d) > \mathrm{R}(P)(D(P,d) - d) \tag{5.2}$$

para algum $\sigma_k \in P$. Existem dois casos a serem considerados:

Caso 1. Um job de σ_k é liberado antes do instante d e possui *deadline* $D(\sigma_k,t) > d$. Como $\phi_k(P,t)$ é a demanda máxima a ser executada no intervalo $[t, D(\sigma_k,t))$ e d é o próximo *deadline* após t, os novos jobs que chegarem no instante d e que poderiam interferir na execução de σ_k já terão sido considerados por $\phi_k(P,t)$. Portanto, pode-se afirmar que o valor de $\phi_k(P,d)$ é igual a $\phi_k(P,t)$ descontado o que foi executado entre t and d. Ou seja,

$$\phi_k(P,d) = \phi_k(P,t) - \rho^*(P,d)(d-t) \tag{5.3}$$

Como $\psi_k(P,t) = \mathrm{R}(P)(D(\sigma_k,t) - d)$, $\psi_k(P,d) = \mathrm{R}(P)(D(\sigma_k,d) - D(P,d))$ e, por hipótese para este caso, $D(\sigma_k,d) = D(\sigma_k,t)$, pode-se concluir que

$$\psi_k(P,t) - \psi_k(P,d) = \mathrm{R}(P)(D(P,d) - d) \tag{5.4}$$

Das inequações (5.2) e (5.4), temos

$$\phi_k(P,d) - \psi_k(P,d) > \psi_k(P,t) - \psi_k(P,d)$$

levando a

$$\phi_k(P,d) - \psi_k(P,t) > 0$$

Substituindo o termo ao lado direito de (5.3) por $\phi_k(P,d)$ nesta última inequação, chegamos a

$$\phi_k(P,t) - \psi_k(P,t) > \rho^*(P,d)(d-t)$$

o que contradiz a Equação (5.1).

Caso 2. Um job de σ_k é liberado em $d = D(\sigma_k,t)$. Neste caso, todo o trabalho $\phi_k(P,t) \geq 0$ é executado até d, já que σ_k libera um novo job no instante d. Além disso, sabe-se que $\rho^*(P,t) \leq \mathrm{R}(P)$. Sejam J_1 e J_2 dois jobs consecutivos de σ_k, o primeiro ativo em t e o segundo liberado em d. Por isso, pode-se considerar J_2 como sendo liberado em t ao invés de d mas mantendo o seu *deadline* em $D(\sigma_k,d)$. Além disso, J_1 é modificado através da adição de $(\mathrm{R}(P) - \rho^*(P,t))(d-t)$ ao seu custo de execução. Estas alterações nos jobs de σ_k não modificam nem o valor de $\rho^*(P,d)$ nem as decisões de escalonamento realizadas após d, embora a inflação de J_1 aumente o valor de $\rho^*(P,t)$ para $\mathrm{R}(P)$. Através das modificações dos jobs de σ_k, o caso 2 é transformado no caso 1 e chega-se ao resultado desejado.

A análise destes dois casos mostram que $\rho^*(P,d) \leq \mathrm{R}(P)$ para qualquer intervalo $[t,d)$. Como resultado, os servidores QPS são capazes de escalonar P considerando o cenário artificialmente produzido. Como o trabalho restante em t que deve ser concluído até d, incluindo o acréscimo dos jobs artificiais, é factível com uma taxa de execução $\rho^*(P,t) \leq \mathrm{R}(P)$, a remoção destes jobs artificiais manterá a carga de trabalho atual como escalonável por EDF. Isto pode ser afirmado pois os *deadlines* dos jobs artificiais é d e pela Definição 5.3.1, $\mathrm{R}(P) - \mathrm{R}(\sigma) < 1$ para qualquer servidor em P.

Para concluir, pode-se observar que este lema igualmente procede quando QPS está configurado para utilizar as estratégias de adaptação CF e RF, pois estas estratégias utilizam $\mathrm{R}(P) \geq \rho(P,t)$ para configurar as taxas dos servidores QPS. □

Finalmente, é possível provar a correção do QPS.

Teorema 5.5.1. *Seja Γ um conjunto de tarefas esporádicas com deadlines implícitos a serem escalonadas em $m = \lceil \mathrm{R}(\Gamma) \rceil$ processadores idênticos. A política de escalonamento QPS produz*

um escalonamento válido para Γ.

Demonstração. De acordo com o Lema 5.4.1, qualquer sistema de tarefas com *deadlines* implícitos e com taxa não superior a m pode ter suas tarefas alocadas a um conjunto de *execution sets* associados com um máximo de m processadores pelo Algoritmo 5.1. É preciso mostrar que nenhuma tarefa em Γ deixará de cumprir seus requisitos temporais. Em verdade, será mostrado um resultado um pouco mais geral, em que qualquer servidor (seja ele uma tarefa ou não) cumpre todos os seus *deadlines*. Primeiro será considerado o caso em que todas as tarefas em Γ sempre estão ativas (que equivale ao caso periódico), e depois este resultado será estendido de forma a considerar que as tarefas podem permanecer inativas durante algum intervalo de tempo (caso esporádico).

Apenas tarefas periódicas. No caso de *minor execution sets*, este resultado é imediato pois cada *minor execution set* é alocado em um único processador dedicado pelas linhas 13-15 do Algoritmo 5.1 e QPS utiliza EDF local para escaloná-los, sendo EDF um algoritmo ótimo para o escalonamento de sistemas com um único processador [Baruah et al. 1990b].

Considere um *major execution set* P alocado ao processador j. Note que o último processador alocado pelo Algorithm 5.1 contém um *minor execution set* e, portanto, deve existir ao menos um processador associado a um *minor execution set*. Assim, procede-se indutivamente em ordem decrescente de processador, considerando para o escalonamento de P no processador j que o escalonamento dos servidores nos processadores $m, \ldots, j+1$ ocorreu de forma correta, sendo os processadores escalonados por EDF no caso base.

Sejam σ_j^A, σ_j^B, e σ_j^S os servidores QPS encarregados do escalonamento de P. A hipótese indutiva utilizada garante que os servidores executados nos processadores de $j+1$ até m foram escalonados corretamente e σ_j^M está associado a algum destes processadores, portanto os jobs de σ_j^M atenderam aos seus *deadlines*. O Algoritmo 5.3 escalona σ_j^S sempre que σ_j^M executa e, portanto, os jobs de σ_j^S também cumprem todos os seus *deadlines*. O restante do tempo do processador j é ocupado com a execução de σ_j^A e de σ_j^B que, como compartilham os mesmos *deadlines* com σ_j^S, atendem aos seus requisitos temporais.

Tarefas esporádicas. Considere que algumas tarefas em Γ estejam inativas durante algum intervalo de tempo. No caso de todas estas tarefas estiverem associadas com *minor execution sets*, estes *minor execution sets* continuarão a ser escalonadas por EDF em seus processadores dedicados e nada precisa ser provado. Caso contrário, estas tarefas inativas provocam uma mudança de modo em pelo menos um *major execution set* P. Sabe-se, pelo Lema 5.5.2, que as mudanças de modo, gerenciadas pelo Algoritmo 5.2 e de acordo com as estratégias de adaptação CF, RF e RP, podem ser aplicadas aos servidores em P com segurança. Além disso, o Algoritmo 5.2 não ultrapassa a taxa de execução máxima definida para o servidor externo pela linha 7 do Algoritmo 5.1 quando da ativação do servidor *master* encarregado das tarefas em P. Portanto, a mudança de modo não irá comprometer a escalonabilidade das tarefas que executam no mesmo processador de P, concluindo a prova. □

5.6 AVALIAÇÃO DO QPS

O desempenho do QPS em termos da quantidade de preempções e de migrações provocadas sobre as tarefas do sistema foi avaliado através de simulações. Foram gerados conjuntos de tarefas de acordo com o procedimento descrito por Emberson *et al.* [Emberson et al. 2010]. As taxas das tarefas geradas foram uniformemente distribuídas no intervalo $(0, 1)$ com seus períodos inteiros distribuídos uniformemente em $[1, 100]$. Para cada simulação, foram considerados 1.000 conjuntos de tarefas, e cada simulação executou por 1.000 unidades de tempo. Como o procedimento de geração de tarefas tende a produzir tarefas com pequena utilização em conjuntos com grande quantidade de tarefas, e isso favoreceria QPS devido a maior probabilidade de particionamento, estes conjuntos foram gerados com não mais do que $4m$ tarefas, com m variando entre 2 e 32 processadores.

Como QPS induz uma hierarquia entre os processadores ao quasi-particionar o sistema de tarefas, este aspecto é avaliado na Seção 5.6.1. Na Seção 5.6.2 são discutidos os efeitos provocados pelas diferentes estratégias de quasi-particionamento no desempenho de QPS. A intenção não é a de avaliar estas estratégias de quasi-particionamento, mas sim destacar quais características destas heurísticas afetam o desempenho de QPS. As Seções 5.6.3 e 5.6.4 comparam QPS com outros algoritmos de escalonamento ótimos para sistemas de tarefas periódicas e esporádicas, respectivamente.

5.6.1 Hierarquia de Processadores

Um importante aspecto a ser analisado com relação ao QPS diz respeito a hierarquia gerada entre os processadores. Como os servidores externos podem fazer parte de outros *major execution sets* e como mudanças de modo não ocorrem em sistemas com tarefas periódicas, quanto mais níveis houver na hierarquia de processadores, maior será o *overhead* devido às preempções e migrações geradas nestes sistemas.

Se for forçado o estabelecimento de uma hierarquia de processadores que possua a maior altura possível, no pior caso um *major execution set* poderá ser produzido no primeiro processador e propagar a formação de outros *major execution sets* por todos os demais processadores até que o último seja atingido. Neste caso, o quasi-particionamento formará uma cadeia de processadores com a maior altura possível. Por exemplo, o quasi-particionamento de um sistema com 10 tarefas com taxa igual a $0,9$ e que devem ser escalonadas em 9 processadores forma uma cadeia de processadores com 8 *major execution sets* e um *minor execution set*. Assim, as tarefas pertencentes ao j-ésimo processador da cadeia podem migrar para qualquer dos $m - j$ processadores localizados hierarquicamente acima dele na cadeia.

De maneira inversa, se uma tarefa esporádica pertencente ao primeiro *execution set* torna-se inativa durante algum tempo, todos os servidores *master* hierarquicamente superiores tornam-se inativos, toda a cadeia de processadores irá se desfazer e o sistema poderá ser escalonado por EDF local. Assim, para sistemas com tarefas esporádicas é esperado que uma hierarquia

de processadores mais vertical produza melhores resultados em termos de preempções e de migrações quando tarefas se tornam inativas por algum tempo. Isto deve-se ao fato de uma quantidade maior de *execution sets* se encontrar no modo EDF devido a inatividade de alguma tarefa do sistema. Em outras palavras, desempenho de QPS é uma função da hierarquia de processadores produzida durante as fases de quasi-particionamento e de alocação dos *execution sets* pelos processadores.

Nesta seção será apresentado o tamanho médio das hierarquias geradas para os conjuntos de tarefas simulados, sendo este cálculo ponderado pela quantidade de processadores em cada nível da hierarquia. Para o exemplo com 9 processadores fornecido há pouco, o tamanho obtido para a hierarquia seria $\frac{1}{9}(0 + 1 + \cdots + 8) = 4$.

A Figura 5.9 resume os tamanhos médios encontrados para os conjuntos de tarefas simulados, seguindo a estratégia de quasi-particionamento FFD. Os resultados obtidos com relação ao tamanho das hierarquias para a estratégia EFD foram semelhantes e por esta razão foram omitidos. Cada conjunto possui taxa total igual a m, de forma a produzirem grandes hierarquias. Como esperado, para sistemas com $m+1$ tarefas, o tamanho médio das hierarquias encontradas foi $\frac{m-1}{2}$. Curiosamente, este valor se reduz abruptamente para conjuntos maiores.

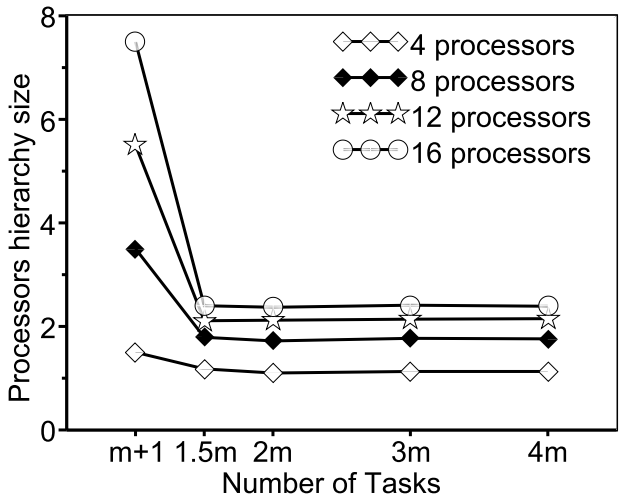


Figura 5.9. Média de hierarquia de processadores em sistemas que utilizam 100% de m processadores.

5.6.2 Heurísticas de quasi-particionamento

O histograma exibido na Figura 5.10 mostra a distribuição dos excessos utilizados para configurar os servidores externos pelo Algoritmo 5.1 (linha 7) para sistemas com 16 tarefas e 8 processadores. Estes valores representam a maior taxa permitida para servidores *master/slave*, ativados pelo Algoritmo 5.2 e, portanto, têm interferência significativa no desempenho de QPS. De fato, quanto menor for a taxa de um servidor *master* melhor QPS lidará com tarefas esporádicas, já que os *major execution sets* estarão sendo executados mais frequentemente no modo

EDF. Neste contexto, como pode ser visto no gráfico da figura citada, FFD tende a favorecer o escalonamento de tarefas esporádicas: aproximadamente 80% dos *execution sets* gerados por FFD não possuem excesso que ultrapassem 5%; EFD distribui os excessos gerados mais igualitariamente entre os *execution sets*.

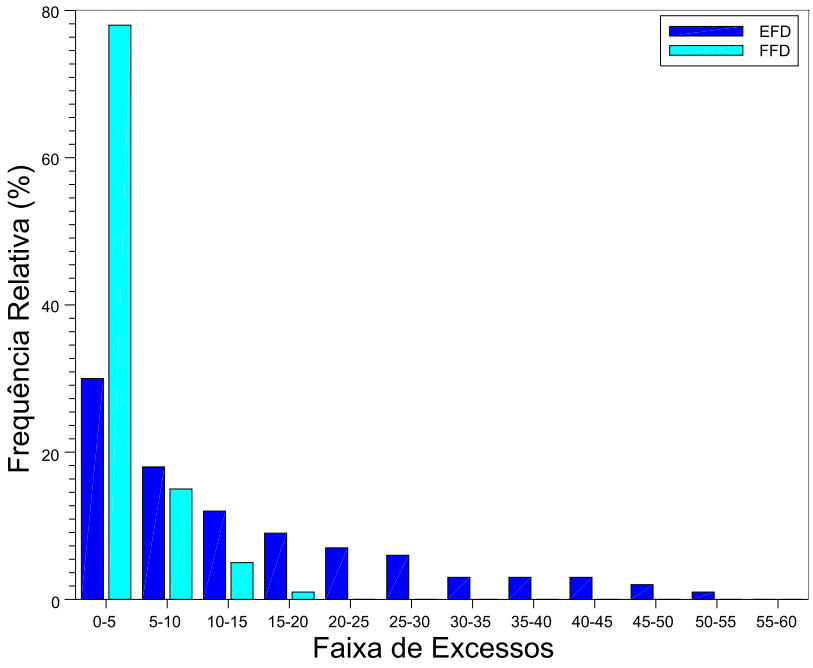


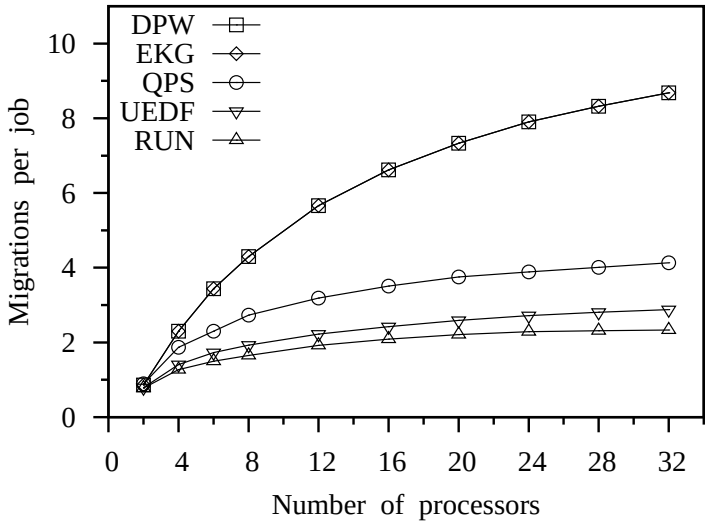
Figura 5.10. Distribuição dos excessos em *execution sets* de acordo com a política de empacotamento para conjuntos com 16 tarefas escalonadas em 8 processadores.

De acordo com o exposto em relação ao tamanho dos excessos produzidos nos *execution sets* e com relação ao tamanho da hierarquia de processadores, seria esperado que o algoritmo FFD favorecesse aos sistemas com tarefas esporádicas, enquanto EFD produzisse melhores resultados para sistemas com tarefas periódicas. Esta expectativa se confirmou. Nas próximas seções serão apresentados os resultados obtidos por QPS para sistemas com tarefas periódicas e esporádicas, utilizando respectivamente os algoritmos de quasi-particionamento EFD e FFD.

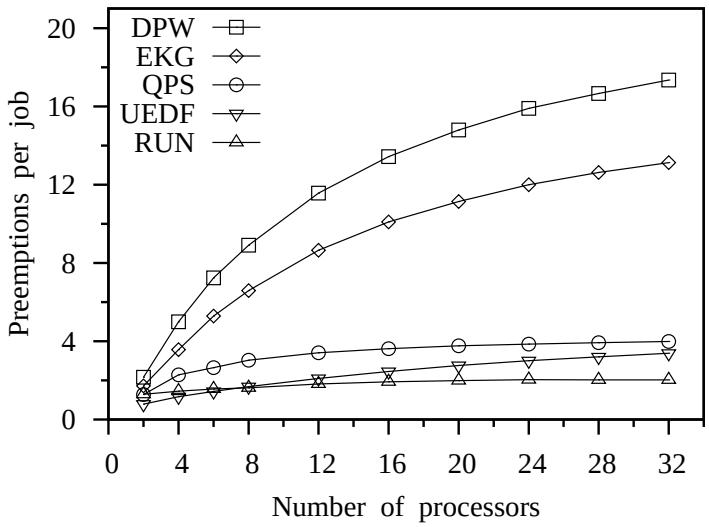
5.6.3 Desempenho para sistemas com tarefas periódicas

A Figura 5.11 compara o desempenho de QPS (utilizando a heurística de quasi-particionamento EFD) com outros algoritmos ótimos para sistemas com tarefas periódicas, no escalonamento de sistemas que utilizam 100% dos processadores envolvidos. Cada ponto no gráfico corresponde à média dos resultados para 1.000 conjuntos de tarefas. Observe que, como DP-Wrap (DPW) [Levin et al. 2010] e EKG [Andersson e Tovar 2006] forçam a igualdade dos *deadlines* para atingir a otimalidade, eles tendem a ter um desempenho inferior a QPS, RUN [Regnier et al. 2011] e U-EDF [Nelissen et al. 2012], que utilizam estratégias diferentes. No geral, RUN

apresenta os melhores resultados para sistemas periódicos, enquanto o desempenho de U-EDF se encontra entre os desempenhos obtidos por RUN e QPS. Vale ressaltar que, como todos os servidores estão sempre ativos (pois as tarefas são periódicas), as estratégias de adaptação de QPS nunca são utilizadas. Mesmo assim, QPS consegue bons resultados em comparação com os melhores algoritmos conhecidos até o presente momento para sistemas com tarefas periódicas.



(a) migrações



(b) preempções

Figura 5.11. Quantidade média de preempções e de migrações para sistemas periódicos com $2m$ tarefas que utilizam totalmente os processadores.

RUN e U-EDF são os melhores algoritmos de escalonamento com otimalidade comprovada para sistemas com tarefas periódicas e esporádicas, respectivamente, e por esta razão serão adotados como base de comparação nas avaliações do QPS.

A Figura 5.12 mostra o comportamento de QPS comparado com RUN e com U-EDF para sistemas periódicos com diferentes quantidades de processadores e de tarefas. Novamente,

todos os sistemas considerados utilizaram 100% dos m processadores. Como pode ser visto nas Figuras 5.12a e 5.12c, o desempenho de QPS para $m + 1$ tarefas não é tão bom como para outros cenários. Isto é esperado pelo fato de sistemas com $m + 1$ tarefas utilizando totalmente m processadores produzirem hierarquias de processadores com alto tamanho (Seção 5.6.1). A performance de QPS melhora significativamente quando são considerados conjuntos com maior quantidade de tarefas. Nas figuras, são apresentados apenas sistemas com $2m$ tarefas, pois o comportamento de conjuntos com maior quantidade de tarefas não se altera significativamente. Vale observar nestes experimentos que, uma vez que todos os servidores estão sempre ativos (pois as tarefas são periódicas), a estratégia de adaptação utilizada pelo QPS não interfere nos resultados.

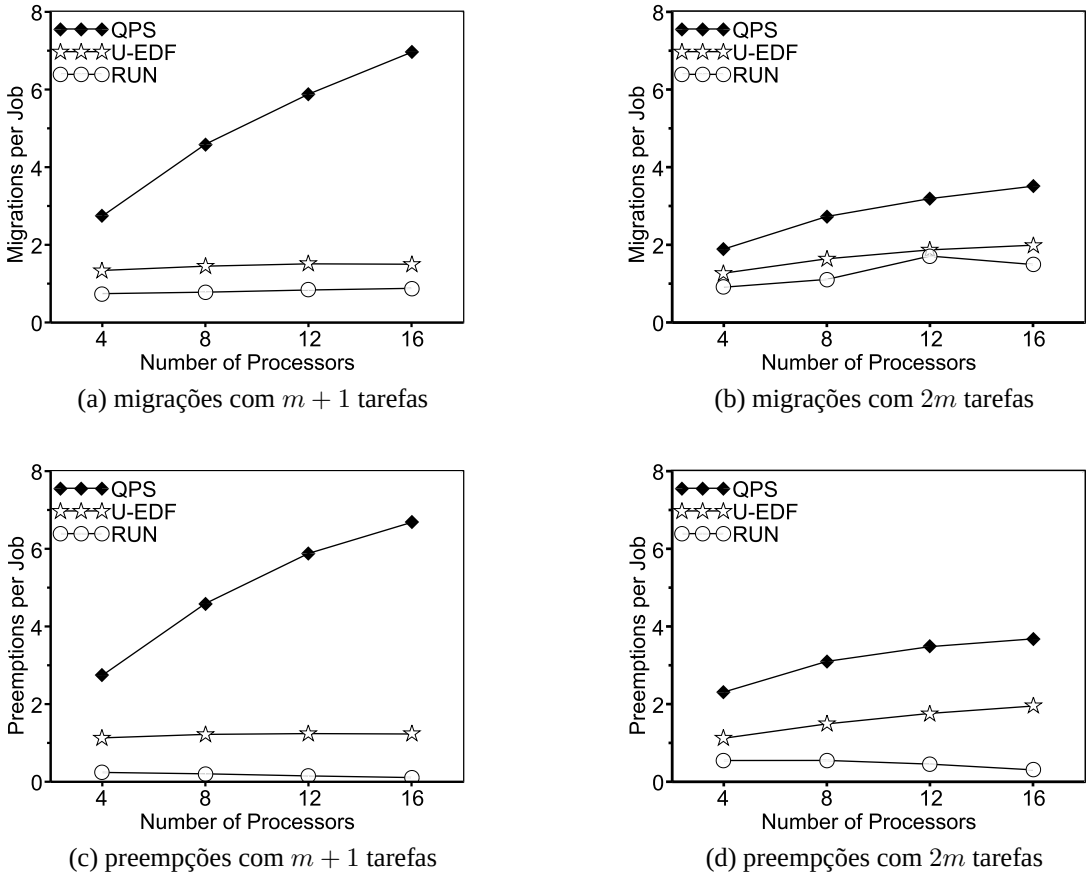
De forma interessante, quando os experimentos são realizados com sistemas que não requerem 100% dos processadores, o desempenho de QPS melhora significativamente equiparando-se em alguns casos aos resultados obtidos por RUN. Isto é ilustrado pela Figura 5.13, que apresenta resultados obtidos no escalonamento de sistemas que utilizam 98% de m processadores. A melhora apresentada é creditada à redução ocorrida no tamanho da hierarquia de processadores. Após o quasi-particionamento dos sistemas, o maior tamanho de hierarquia encontrado foi igual a 2. Como pode ser visto, a melhora no desempenho do U-EDF neste mesmo cenário (Figuras 5.12 e 5.13) não chega a ser significativa, indicando que o U-EDF como política global não se aproveita de um possível particionamento do sistema de tarefas, como o QPS consegue fazer.

5.6.4 Desempenho para sistemas com tarefas esporádicas

A grande vantagem do uso do algoritmo QPS vem quando são considerados sistemas com tarefas esporádicas, como torna-se evidente nesta seção. Como RUN não é capaz de escalonar tarefas esporádicas, a avaliação do desempenho de QPS (utilizando a heurística de quasi-particionamento FFD) no cenário formado por sistemas com tarefas esporádicas será feita apenas através da sua comparação com U-EDF.

Durante os experimentos pode-se observar que o desempenho de QPS não varia significativamente em função da quantidade de processadores do sistema. O padrão encontrado é muito similar ao encontrado na Figura 5.14, que apresenta os resultados para $m = 16$ processadores. É interessante observar que sistemas com $m + 1$ tarefas apresentam menos migrações do que sistemas com $1.5m$ tarefas. A exceção ocorre quando o atraso é baixo, até 10 unidades de tempo. A explicação para isto está relacionada com o tamanho da hierarquia de processadores. Como observado na Seção 5.6.1, sistemas configurados com $m + 1$ tarefas tendem a produzir uma cadeia que se quebra quando as tarefas dos processadores hierarquicamente mais abaixo tornam-se inativas. Todavia, quando o atraso é baixo, o sistema se comporta próximo a um sistema periódico e $m + 1$ tarefas causam uma maior quantidade de preempções e migrações.

A Figura 5.15 ilustra bem a forma como QPS converge de um escalonamento global para



(a) migrações com $m + 1$ tarefas

(b) migrações com $2m$ tarefas

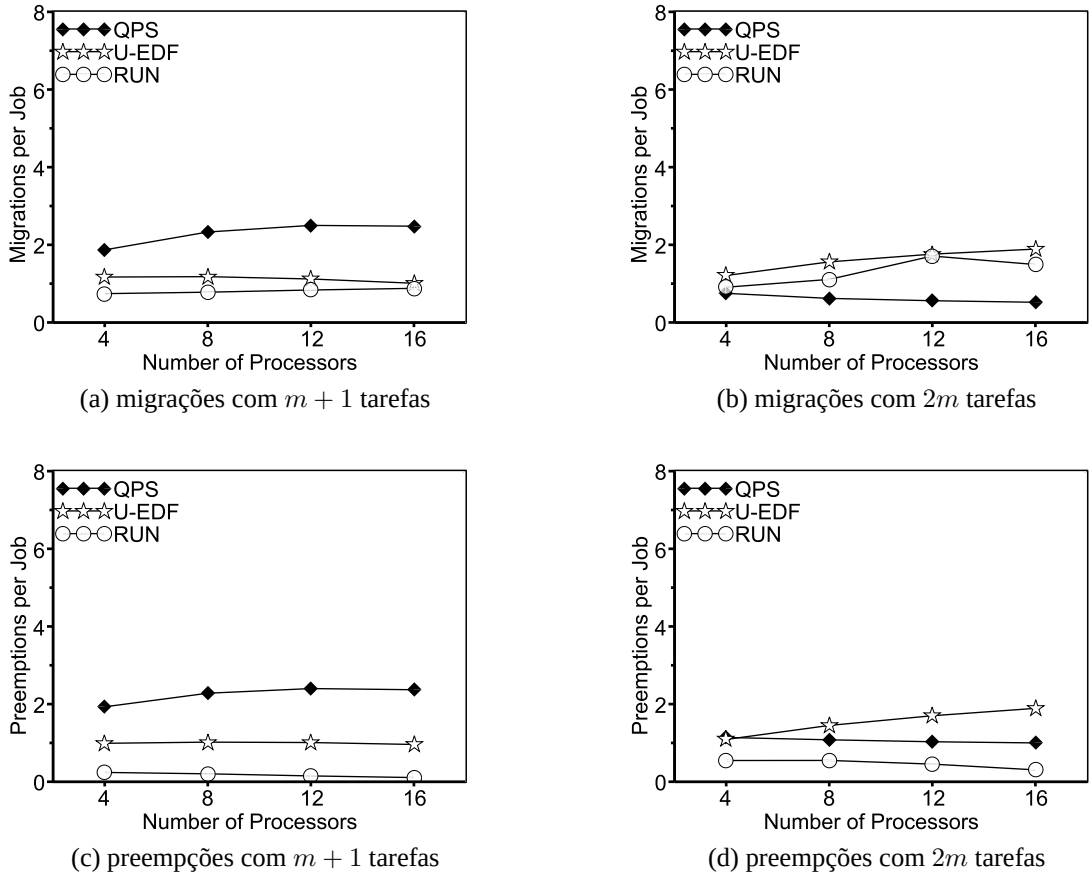
(c) preempções com $m + 1$ tarefas

(d) preempções com $2m$ tarefas

Figura 5.12. Quantidade média de migrações e preempções para sistemas periódicos que utilizam totalmente m processadores.

um escalonamento particionado (com migrações residuais) conforme o atraso das tarefas cresce. O eixo horizontal representa o maior atraso que cada tarefa pode experimentar durante a simulação. Mais especificamente, quando uma tarefa é ativada, o simulador produz um atraso com duração uniformemente distribuída no intervalo $[1, 100]$. Caso nenhum atraso seja gerado, os sistemas simulados utilizarão completamente $m = 8$ processadores.

Pode ser visto que QPS converge muito rapidamente para um sistema particionado quando o atraso médio das tarefas cresce, o que acontece em todas as três estratégias de adaptação. Como esperado, as estratégia RF e RP produziram melhores resultados, já que utilizam um cálculo exato para decidir quando deve ser ativado o modo QPS ou quando pode-se simplesmente utilizar o escalonamento EDF. Pode ser observado também que este mesmo comportamento não ocorre com o U-EDF [Nelissen et al. 2012], conforme seria esperado. Embora os resultados para CF sejam piores, é importante ressaltar que a sua implementação é simples. Além disso, de forma semelhante ao U-EDF, o desempenho de QPS não sofre significativa variação em função dos atrasos nas chegadas das tarefas.



(a) migrações com $m + 1$ tarefas

(b) migrações com $2m$ tarefas

(c) preempções com $m + 1$ tarefas

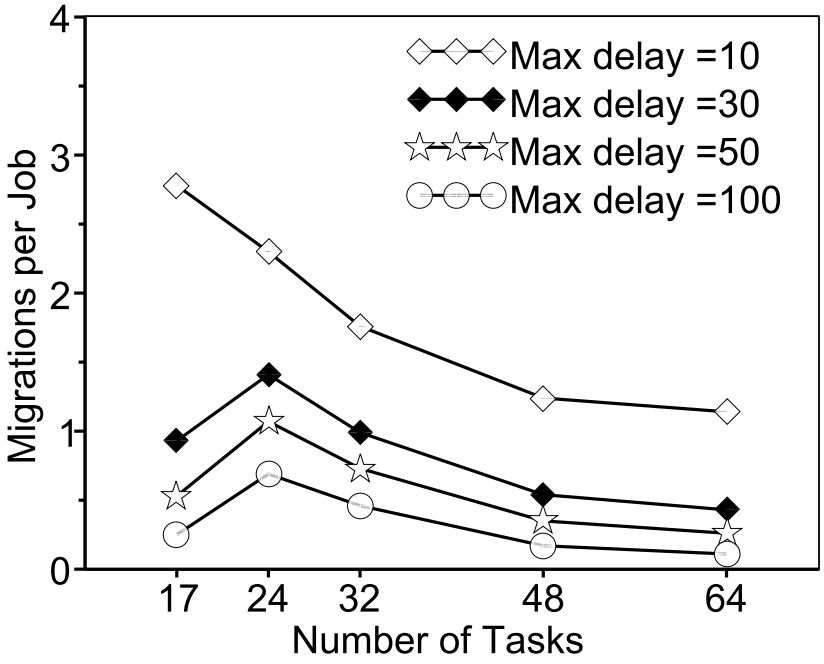
(d) preempções com $2m$ tarefas

Figura 5.13. Quantidade média de migrações e preempções para sistemas periódicos que utilizam 98% de m processadores.

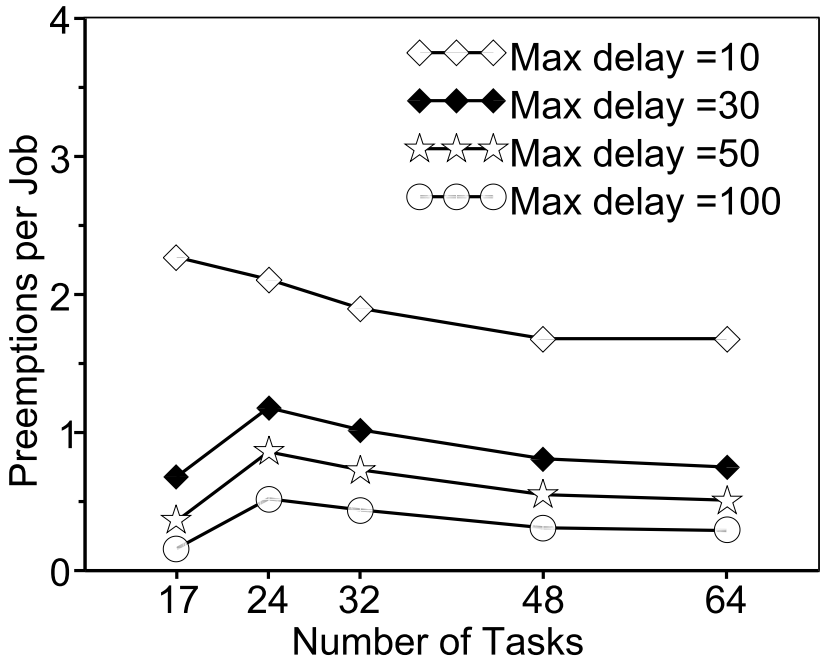
RESUMO

O quasi-particionamento de um conjunto de tarefas é um particionamento deste conjunto que possui a propriedade de, na ausência de uma tarefa de uma partição qualquer, o conjunto de tarefas restante nesta partição ser escalonável por EDF em um único processador. Esta propriedade permite ao QPS adotar uma estratégia adaptativa. Enquanto as tarefas chegam periodicamente, QPS controla a sua execução com um conjunto de servidores de Taxa-Fixa (Seção 3.2) garantindo que apenas tarefas distintas executem simultaneamente em processadores diferentes. Ao ocorrer atraso na chegada de alguma tarefa esporádica, QPS altera o seu comportamento para adotar um escalonamento particionado.

QPS é um algoritmo ótimo no sentido em que produz um escalonamento válido em m processadores para qualquer sistema de tarefas esporádicas com *deadline* implícito que tenha utilização total não superior a m. Além disso, os experimentos realizados mostram que QPS reduz consideravelmente a quantidade de migrações e preempções comparado com os trabalhos relacionados.

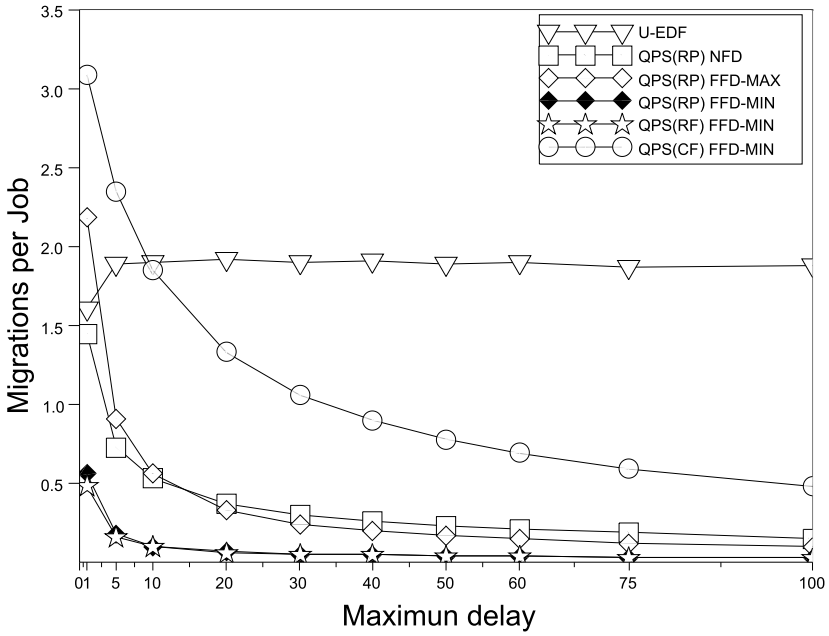


(a) migrações

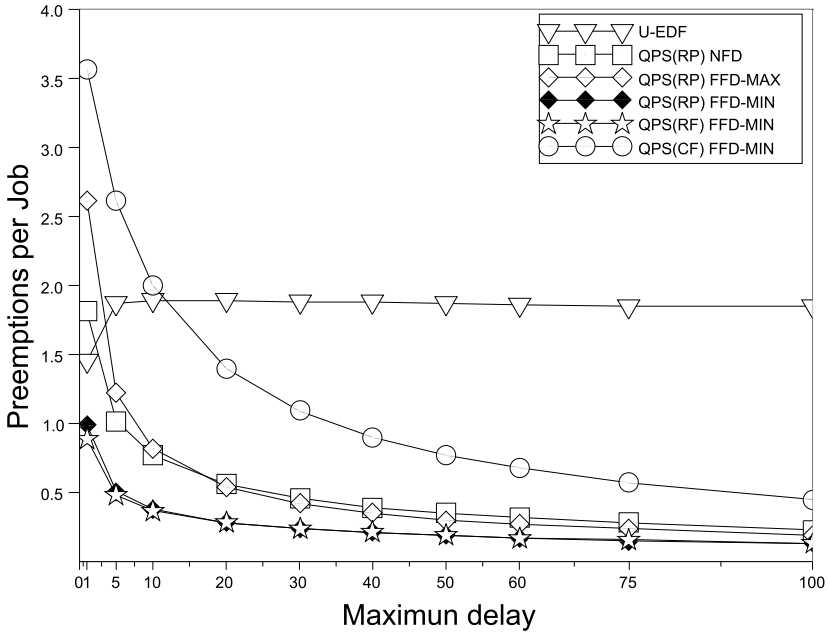


(b) preempções

Figura 5.14. Quantidade média de preempções e migrações para sistemas de tarefas esporádicas com $m = 16$ processadores e atraso máximo de cada tarefa compreendido entre 10 e 100.



(a) migrações



(b) preempções

Figura 5.15. Quantidade média de migrações e preempções para sistemas com 16 tarefas esporádicas escalonadas em 8 processadores.

Capítulo 6

Todos os dias quando acordo,
Não tenho mais o tempo que passou
Mas tenho muito tempo
Temos todo o tempo do mundo

Legião Urbana, Tempo Perdido

CONCLUSÃO

O problema de escalonar sistemas de tempo real em múltiplos processadores com tarefas críticas tem tradicionalmente três possíveis abordagens. Enquanto a abordagem particionada pode fazer uso dos resultados consolidados para ambientes com um único processador mas enfrenta a limitação de ter que distribuir as tarefas adequadamente pelos processadores, o que é um problema NP-Difícil, algoritmos que adotam uma abordagem global conseguem produzir escalonamentos válidos ótimos, mas ao custo de excessivo número de preempções e migrações. A abordagem semi-particionada busca conciliar as duas abordagens anteriores, com o objetivo de reunir as propriedades vantajosas de cada estratégia.

O grande desafio dos algoritmos que seguem a abordagem semi-particionada encontra-se em evitar, sem prejuízo de desempenho, que duas partes de uma mesma tarefa sejam executadas em paralelo em processadores distintos. Neste trabalho, comprova-se que o controle de execução das tarefas migratórias pode ser feito com a utilização de servidores, sendo apresentados dois novos algoritmos de escalonamento que se utilizam de dois novos tipos de servidores. O primeiro algoritmo, denominado EDF-BR, aloca tarefas esporádicas estaticamente aos processadores, permitindo que algumas delas migrem entre processadores. Para o controle da execução das tarefas, EDF-BR divide o fluxo do tempo em janelas de igual duração, fazendo uso de servidores BR que executam em intervalos regulares de tempo.

Segundo algoritmo desenvolvido, QPS é um algoritmo ótimo, capaz de produzir um escalonamento válido em m processadores para qualquer conjunto de tarefas esporádicas com *deadline* implícito que possua utilização total não superior a m. QPS introduz o conceito de quasi-partição e o utiliza para adaptar a política de escalonamento à carga do sistema em tempo

de execução. Este algoritmo flutua entre a utilização de servidores de Taxa-Fixa, quando as tarefas do sistema têm comportamento próximo ao das tarefas periódicas, e a utilização de EDF particionado, quando as tarefas do sistema atrasam com maior frequência. Esta possibilidade de adaptação à carga do sistema em tempo possibilita uma redução acentuada da quantidade de preempções e de migrações, comparados ao estado da arte.

Os resultados obtidos por este estudo levantam algumas possibilidades interessantes de pesquisa, proporcionando possíveis trabalhos futuros. Alguns destes são apresentados a seguir:

- Aplicações multimídia, apesar de serem usualmente caracterizadas como tarefas não críticas, podem coexistir com tarefas críticas de tempo real. Um exemplo desta situação é a utilização de realidade aumentada no auxílio de cirurgias [Kersten-Oertel et al. 2013]. QPS pode ser utilizado para o gerenciamento da execução das tarefas críticas em multiprocessadores, assim como servidores QPS podem prover o encapsulamento das tarefas não críticas, permitindo a coexistência destas duas classes de tarefas em um mesmo sistema.
- Apesar dos modelos usualmente utilizados nas pesquisas sobre escalonamento de processos considerarem as tarefas independentes entre si, sistemas reais são compostos por tarefas que possuem interdependência tanto com relação à sua ordem de execução como com relação ao compartilhamento de outros recursos além dos processadores. O empacotamento de tarefas interdependentes em um mesmo servidor QPS pode ser uma estratégia interessante para impor uma ordenação necessária às tarefas do sistema.
- Este trabalho foi realizado tendo como uma de suas motivações a proliferação das arquiteturas com múltiplos processadores. *Networks-on-Chip* (NoC) constitui-se em uma arquitetura que interconecta de centenas a milhares de processadores em um único chip [Radetzki et al. 2013], o que somente é possível devido à tendência atual de miniaturalização dos circuitos. Esta tendência, entretanto, aumenta a probabilidade de ocorrência de falhas. O controle da execução de tarefas em paralelo, exercido por QPS, pode ser explorado para prover a redundância necessária às técnicas de tolerância à falhas que tornariam mais confiáveis os sistemas que fossem executados sobre esta arquitetura.

Os algoritmos propostos nesta tese oferecem uma nova perspectiva sobre como gerenciar o escalonamento de tarefas esporádicas críticas em múltiplos processadores. O fato de um deles, o algoritmo QPS, ter as características de ser ótimo, não baseado no conceito de *Proportionate Fairness* e adaptável dinamicamente à carga de trabalho do sistema, permite a sua implementação em sistemas reais, abrindo um novo horizonte sobre o problema do escalonamento em multiprocessadores.

REFERÊNCIAS BIBLIOGRÁFICAS

ABENI, L.; BUTTAZZO, G. Integrating multimedia applications in hard real-time systems. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 1998. p. 4–13.

ANDERSON, J.; SRINIVASAN, A. Early-release fair scheduling. In: *Proceedings of IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. [S.l.: s.n.], 2000. p. 35–43. ISSN 1068-3070.

ANDERSSON, B.; BLETSAS, K. Sporadic multiprocessor scheduling with few preemptions. In: *Proceedings of IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. [S.l.: s.n.], 2008. p. 243–252.

ANDERSSON, B.; BLETSAS, K.; BARUAH, S. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2008. p. 385–394.

ANDERSSON, B.; JONSSON, J. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In: *Work-in-Progress Session: Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2000. p. 53–56.

ANDERSSON, B.; JONSSON, J. Preemptive multiprocessor scheduling anomalies. In: *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. [S.l.: s.n.], 2002. p. 12–19.

ANDERSSON, B.; TOVAR, E. Multiprocessor scheduling with few preemptions. In: *Proceedings of IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*. [S.l.: s.n.], 2006. p. 322–334.

AUDSLEY, N. C.; BURNS, A.; RICHARDSON, M.; TINDELL, K.; WELLINGS, A. J. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 1993. v. 8, n. 5, p. 284–292, 1993.

BAKER, T. P. Multiprocessor edf and deadline monotonic schedulability analysis. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2003. p. 120. ISBN 0-7695-2044-8.

BAKER, T. P.; CIRINEI, M.; BERTOGNA, M. Edzl scheduling analysis. *Real-Time Systems*, 2008. Kluwer Academic Publishers, v. 40, n. 3, p. 264–289, 2008.

BARUAH, S.; BAKER, T. Schedulability analysis of global edf. *Real-Time Systems*, 2008. Kluwer Academic Publishers, Norwell, MA, USA, v. 38, n. 3, p. 223–235, 2008. ISSN 0922-6443.

BARUAH, S.; CHEN, D.; GORINSKY, S.; MOK, A. Generalized multiframe tasks. *Real-Time Systems*, 1999. Springer Netherlands, v. 17, p. 5–22, 1999. ISSN 0922-6443. 10.1023/A:1008030427220. Disponível em: <http://dx.doi.org/10.1023/A:1008030427220>.

BARUAH, S.; COHEN, N. K.; PLAXTON, C. G.; VARVEL, D. A. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 1996. v. 15, n. 6, p. 600–625, 1996.

BARUAH, S.; GEHRKE, J.; PLAXTON, C. Fast scheduling of periodic tasks on multiple resources. In: *Proceedings of the International Parallel Processing Symposium*. [S.l.: s.n.], 1995. p. 280–288.

BARUAH, S.; GOOSSENS, J. Scheduling real-time tasks: Algorithms and complexity. In: LEUNG, J. Y.-T. (Ed.). *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. [S.l.]: Chapman Hall/CRC Press, 2004.

BARUAH, S.; MOK, A.; ROSIER, L. Preemptively scheduling hard-real-time sporadic tasks on one processor. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 1990b. p. 182 –190.

BARUAH, S.; ROSIER, L.; HOWELL, R. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 1990a. Kluwer Academic Publishers, v. 2, n. 4, p. 301–324, 1990a. ISSN 0922-6443. Disponível em: <http://dx.doi.org/10.1007/BF01995675>.

BARUAH, S. K.; HOWELL, R. R.; ROSIER, L. E. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 1993. v. 118, n. 1, p. 3 – 20, 1993. ISSN 0304-3975. Disponível em: <http://www.sciencedirect.com/science/article/pii/0304397593903606>.

BLETSAS, K.; ANDERSSON, B. Notional processors: An approach for multiprocessor scheduling. In: *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. [S.l.: s.n.], 2009. p. 3–12. ISSN 1080-1812.

BURNS, A. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 1991. IEE, v. 6, p. 116–128(12), May 1991. ISSN 0268-6961. Disponível em: <http://digital-library.theiet.org/content/journals/10.1049/sej.1991.0015>.

BURNS, A.; DAVIS, R.; WANG, P.; ZHANG, F. Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme. *Real-Time Systems*, 2012. Springer US, v. 48, n. 1, p. 3–33, 2012. ISSN 0922-6443. Disponível em: <http://dx.doi.org/10.1007/s11241-011-9126-9>.

CARPENTER, J.; FUNK, S.; HOLMAN, P.; SRINIVASAN, A.; ANDERSON, J.; BARUAH, S. A categorization of real-time multiprocessor scheduling problems and algorithms. In: *Handbook on Scheduling Algorithms, Methods, and Models*. [S.l.]: Chapman Hall/CRC, Boca, 2004.

CHO, H.; RAVINDRAN, B.; JENSEN, E. An optimal real-time scheduling algorithm for multiprocessors. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2006. p. 101–110.

CODD, E. F. Multiprogram scheduling: parts 1 and 2. introduction and theory. *Communications of ACM*, 1960. ACM, New York, NY, USA, v. 3, n. 6, p. 347–350, jun. 1960. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/367297.367317>.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Survey*, 2011. ACM, New York, NY, USA, v. 43, n. 4, p. 35:1–35:44, out. 2011. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1978802.1978814>.

DERTOUZOS, M. L. Control robotics: The procedural control of physical processes. In: *IFIP Congress*. [S.l.: s.n.], 1974. p. 807–813.

DHALL, S. K.; LIU, C. On a real-time scheduling problem. *Operations Research*, 1978. JSTOR, p. 127–140, 1978.

EMBERSON, P.; STAFFORD, R.; DAVIS, R. I. Techniques for the synthesis of multiprocessor tasksets. In: *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. [S.l.: s.n.], 2010. p. 6–11.

FISHER, N. W. *The Multiprocessor Real-Time Scheduling of General Task Systems*. Tese (Doutorado) — University of North Carolina, Chapel Hill, 2007.

GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. [S.l.]: W. H. Freeman and Company, 1979.

GHAZALIE, T.; BAKER, T. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 1995. Kluwer Academic Publishers, v. 9, n. 1, p. 31–67, 1995. ISSN 0922-6443. Disponível em: <http://dx.doi.org/10.1007/BF01094172>.

GOOSSENS, J.; DEVILLERS, R. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. *Real-Time Systems*, 1997. Kluwer Academic Publishers, v. 13, n. 2, p. 107–126, 1997. ISSN 0922-6443. Disponível em: <http://dx.doi.org/10.1023-/A%3A1007980022314>.

GOOSSENS, J.; RICHARD, P.; RICHARD, P.; BRUXELLES, U. L. D. Overview of real-time scheduling problems. In: *Proceedings of the Euro Workshop on Project Management and Scheduling (PMS)*. [S.l.: s.n.], 2004.

HA, R.; W.S.LIU, J. *Validating Timing Constraints in Multiprocessor and Distributed Real-Time Systems*. [S.l.], 1993.

H.ANDERSON, J.; BUD, V.; C.DEVI, U. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In: *Proceedings of IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. [S.l.: s.n.], 2005. p. 199–208.

HOANG, H.; BUTTAZZO, G.; JONSSON, M.; KARLSSON, S. Computing the minimum edf feasible deadline in periodic systems. In: *Proceedings of IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*. [S.l.: s.n.], 2006. p. 125–134. ISSN 1533-2306.

INTEL. Intel's teraflops research chip: Advancing multi-core technology into the tera-scale era. 2012. 2012. Disponível em: <ftp://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf>.

JOSEPH, M.; GOSWAMI, A. Formal description of realtime systems: a review. *Information and Software Technology*, 1989. v. 31, n. 2, p. 67 – 76, 1989. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/0950584989900852>.

KATO, S.; YAMASAKI, N.; ISHIKAWA, Y. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: *Proceedings of IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. [S.l.: s.n.], 2009. p. 249–258.

KERSTEN-OERTEL, M.; JANNIN, P.; COLLINS, D. L. The state of the art of visualization in mixed reality image guided surgery. *Computerized Medical Imaging and Graphics*, 2013. Elsevier, v. 37, n. 2, p. 98–112, 2013.

KOPETZ, H. *Real-time systems: design principles for distributed embedded applications*. [S.l.]: Springer, 2011.

KOREN, G.; AMIR, A.; DAR, E. The power of migration in multi-processor scheduling of real-time systems. In: *ACM-SIAM Symposium on Discrete Algorithms*. [S.l.: s.n.], 1998. (SODA '98), p. 226–235.

LEHOCZKY, J.; RAMOS-THUEL, S. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 1992. p. 110–123.

LEUNG, J. Y.-T.; WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 1982. v. 2, n. 4, p. 237 – 250, 1982. ISSN 0166-5316. Disponível em: <http://www.sciencedirect.com/science/article/pii/0166531682900244>.

LEVIN, G.; FUNK, S.; SADOWSKI, C.; PYE, I.; BRANDT, S. DP-FAIR: a simple model for understanding optimal multiprocessor scheduling. In: *Proceedings of IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. [S.l.: s.n.], 2010. p. 3–13.

LIU, C. L. Scheduling algorithms for multiprogram in a hard real-time environment. *JPL Space Programs Summary*, 1969. II, p. 37–60, 1969.

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

MANACHER, G. K. Production and stabilization of real-time task schedules. *ACM Journal*, 1967. ACM, New York, NY, USA, v. 14, n. 3, p. 439–465, jul. 1967. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/321406.321408>.

MARTI, P.; FUERTES, J.; FOHLER, G.; RAMAMRITHAM, K. Jitter compensation for real-time control systems. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2001. p. 39–48.

MASSA, E.; LIMA, G. A bandwidth reservation strategy for multiprocessor real-time scheduling. In: *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. [S.l.: s.n.], 2010. p. 175 –183.

MCNAUGHTON, R. Scheduling with deadlines and loss functions. *Management Science*, 1959. v. 6, n. 1, p. 1–12, 1959.

MOK, A. K.-L. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. Tese (Doutorado) — Massachusetts Institute of Technology, 1983.

NELISSEN, G.; BERTEN, V.; GOOSSENS, J.; MILOJEVIC, D. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In: *Proceedings of IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*. [S.l.: s.n.], 2011. v. 1, p. 15 –24. ISSN 1533-2306.

NELISSEN, G.; BERTEN, V.; NELIS, V.; GOOSSENS, J.; MILOJEVIC, D. U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: *Proceedings of IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2012. p. 13–23.

PAOLIERI, M.; QUIñONES, E.; CAZORLA, F.; BERNAT, G.; VALERO, M. Hardware support for wcet analysis of hard real-time multicore systems. In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. [S.l.: s.n.], 2009. p. 57–68. ISBN 978-1-60558-526-0.

PIAO, X.; HAN, S.; KIM, H.; PARK, M.; CHO, Y.; CHO, S. Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. In: *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. [S.l.: s.n.], 2006. p. 359–364.

RADETZKI, M.; FENG, C.; ZHAO, X.; JANTSCH, A. Methods for fault tolerance in networks-on-chip. *ACM Comput. Surv.*, 2013. v. 46, n. 1, p. 8:1–8:38, jul. 2013. ISSN 0360-0300.

RAMAMRITHAN, K.; STANKOVIC, J. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 1994. v. 82, n. 1, p. 55–67, 1994. ISSN 0018-9219.

REGNIER, P.; LIMA, G.; MASSA, E.; LEVIN, G.; BRANDT, S. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2011. p. 104 –115. ISSN 1052-8725.

REGNIER, P.; LIMA, G.; MASSA, E.; LEVIN, G.; BRANDT, S. Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach. *Real-Time Systems*, 2012. Springer US, p. 1–39, 2012. ISSN 0922-6443. Disponível em: <http://dx.doi.org/10.1007-/s11241-012-9165-x>.

SANTOS-JR., A.; LIMA, G.; BLETSAS, K. On the processor utilization bound of the c=d scheduling algorithm. 2013. 2013.

SANTOS-JR., A.; LIMA, G.; BLETSAS, K.; KATO, S. Multiprocessor real-time scheduling with a few migrating tasks. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 2013.

SHA, L.; ABDELZAHER, T.; ÅRZéN, K.-E.; CERVIN, A.; BAKER, T.; BURNS, A.; BUTTAZZO, G.; CACCAMO, M.; LEHOCZKY, J.; MOK, A. K. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 2004. Kluwer Academic Publishers, Norwell, MA, USA, v. 28, n. 2-3, p. 101–155, nov. 2004. ISSN 0922-6443. Disponível em: <http://dx.doi-.org/10.1023/B:TIME.0000045315.61234.1e>.

SPRUNT, B.; SHA, L.; LEHOCZKY, J. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1989. Kluwer Academic Publishers, v. 1, n. 1, p. 27–60, 1989. ISSN 0922-6443.

SPURI, M.; BUTTAZZO, G. Efficient aperiodic service under earliest deadline scheduling. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. [S.l.: s.n.], 1994. p. 2–11.

SPURI, M.; BUTTAZZO, G. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 1996. v. 10, n. 2, p. 179–210, 1996.

STANKOVIC, J. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Transactions on Computers*, 1988. v. 21, n. 10, p. 10–19, 1988. ISSN 0018-9162.

STROSNIDER, J.; LEHOCZKY, J.; SHA, L. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 1995. v. 44, n. 1, p. 73–91, 1995. ISSN 0018-9340.

TILERA. Tile-gx8072 processor. 2013. 2013. Disponível em: <tilera.com/sites/default/files-/productbriefs/ TILE-Gx8072_PB041-02.pdf>.

ZHANG, F.; BURNS, A. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 2009. IEEE Computer Society, Los Alamitos, CA, USA, v. 58, n. 9, p. 1250–1258, 2009. ISSN 0018-9340.