Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Provisão de Qualidade de Serviço em Escalonadores para Sistemas Operacionais Embarcados de Tempo-Real.

David Matschulat

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Fabiano Hessel

Porto Alegre
Fevereiro de 2007

Dados Internacionais de Catalogação na Publicação (CIP)

M434p Matschulat, David
Provisão de qualidade de serviço em escalonadores para sistemas operacionais embarcados de tempo-real / David Matschulat. – Porto Alegre, 2007.
72 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS
Orientador: Prof. Dr. Fabiano Hessel

1. Informática. 2. Sistemas Operacionais. 3. Sistemas Embarcados. 4. Escalonador. I. Título.

CDD 005.4

Ficha Catalográfica elaborada pelo Setor de Processamento Técnico da BC-PUCRS

Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada ***"Provisão de Qualidade de Serviço em Escalonadores para Sistemas Operacionais Embarcados de Tempo-Real"***, apresentada por David Matschulat, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais, aprovada em 26/02/2007 pela Comissão Examinadora:

Prof. Dr. Fabiano Passuelo Hessel – Orientador — PPGCC/PUCRS

Prof. Dr. Eduardo Augusto Bezerra — PPGCC/PUCRS

Prof. Dr. Antonio Augusto Medeiros Fröhlich — UFSC

Homologada em 04/06/2007, conforme Ata No. 013/2007 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 16 – sala 106 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Agradecimentos

Dois anos de muito esforço se passaram e é chegada a hora da conclusão de mais um projeto. Todo trabalho de pesquisa envolve idéias, inspiração, motivação e capacitação. Grande parte desses essenciais substantivos são fruto da interação pessoal com amigos, colegas, professores e família. A importância de tais deve ser ressaltada com a devida honra.

Em primeiro lugar, eu gostaria de agradecer ao meu orientador, Professor Doutor Fabiano Hessel, que durante todo esse tempo me apoiou, incentivou e conduziu.

Também gostaria de agradecer aos muitos colegas de aula e de laboratório que sempre mostraram o companheirismo. Em especial gostaria de agradecer ao Sérgio, Ost, Márcio, Cris, Melissa, Edson e Ewerson pelo espírito de equipe, companheirismo e trabalho duro. Agradeço ao professor Fernando Moraes pelas aulas que tornaram meu trabalho possível e ao professor César Marcon pelos *insights* sobre o trabalho.

Também agradeço aos meus pais que deram um indescritível e impagável suporte para que eu tivesse forças para levantar as pedras do caminho e fazer delas paredes que constroem a vida. Agradeço também aos meus avós maternos que, com suor, sempre viveram para que filhos e netos tivessem uma vida melhor.

E, por último, mas com a maior honra, agradeço a Deus pela verdade, amor, força, esperança e salvação.

"Stand ye in the ways, and see, and ask for the old paths, where is the good way, and walk therein, and ye shall find rest for your souls."
(Jer 6:16)

Resumo

Atender requisitos de qualidade de serviço (QoS, do inglês, Quality of Service) em sistemas embarcados como, por exemplo, de multimídia, pode ser realizado de forma fim-a-fim, i.e., de um ponto de geração de dados a um ponto de consumo de dados. A inserção de mecanismos de controle e gerência da qualidade faz-se necessária internamente aos sistemas operacionais (SO), pois SOs têm um importante papel na provisão de QoS fim-a-fim. A implementação de tais mecanismos inclui controle de admissão e reserva de recursos, bem como, o controle de escalonamento de processos e monitoração ativa de QoS entregue. Neste trabalho foram realizados o estudo e a implementação da provisão de QoS em escalonadores para sistemas operacionais embarcados de tempo-real. Baseado em conceitos e análise de trabalhos relacionados, um novo algoritmo de escalonamento, ER-EDF, é proposto. ER-EDF apresenta melhorias de performance e suporte simplificado às tarefas de tempo-real do tipo *hard*.

Palavras-chave: Qualidade de Serviço, Escalonador, Sistemas Embarcados, Tempo-Real, Sistemas Operacionais.

Abstract

Fulfilling Quality of Service (QoS) requirements in embedded systems, e.g., multimedia systems, can be provided in an end-to-end manner, i.e., from a data generator point to a data consumer point. Management and control mechanisms are necessery in operating systems' (OS) internals, for OSs play a important role in end-to-end QoS provision. The implementation of such mechanisms includes admission control and resource reservation, as well as process scheduling control and active monitoring of the delivered QoS. QoS provisioning for embedded real-time operating systems is the main subject of this work, which presents the study and implementation of processor time reservation in an embedded system scheduler. Based on concepts and analysis of related works, a new scheduling algorithm, ER-EDF, is proposed. ER-EDF adds performance and simplified hard real-time support to applications.

Keywords: Quality of Service, Scheduler, Embedded Systems, Real-Time, Operating Systems.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	*Application Program Interface*	30
CD-R	*Compact Disk - Recordable*	22
CI	Circuito Integrado	25
CPU	*Central Processing Unit*	28
EDF	*Earliest Deadline First*	37
EP	Elemento de Processamento	26
GCC	GNU C Compiler	45
GNU	GNU’s Not Unix	45
GPS	*Global Positioning System*	25
MPSoC	*Multiprocessor System on Chip*	22
QoS	Quality of Service	21
SoC	*System on Chip*	25
SOE	Sistema Operacional Embarcado	23
SO	Sistema Operacional	21
SRAM	*Static Random Access Memory*	44

Sumário

1 Introdução

O termo QoS (do inglês, qualidade de serviço) tem sido muito referenciado nos últimos anos, devido, principalmente, a sua larga utilização em ambientes de rede distribuídos, que exigem certos requisitos do sistema, como retardo máximo, variação estatística máxima do retardo (*jitter*), e taxa mínima de transmissão. Oferecer um serviço com QoS significa permitir que tais parâmetros sejam garantidos aos usuários de recursos durante o período de utilização, baseando-se em valores de configuração fornecidos pelos usuários.

Para atender requisitos de QoS em sistemas distribuídos como, por exemplo, de multimídia, é necessário fazê-lo de forma fim-a-fim, i.e., de um ponto de geração de dados a um ponto de consumo de dados. Por exemplo, durante a exibição de um vídeo remoto, a comunicação ocorre entre um cliente e um servidor. Para este serviço, as garantias de QoS devem atuar sobre todo o fluxo de dados: servidor remoto, rede e receptor. Conforme ilustrado na Figura 1, isso requer a provisão de mecanismos de controle e gerência fim-a-fim, o que inclui controle de admissão e reserva de recursos, controle de escalonamento de processos e monitoração ativa da qualidade do serviço entregue [1]. Na Figura 1 estão representados um ponto de origem e um ponto de destino de um fluxo de dados. Os dados passam por cada um dos pontos representados na figura, que devem ser gerenciados e controlados pelo sistema de QoS. A parte do sistema operacional (SO) inicia com os *buffers* de entrada/saída e termina na pilha de protocolos, seguido, então, pela comunicação através da rede e, no receptor, passa pelos estágios do SO até chegar no destino dos dados.

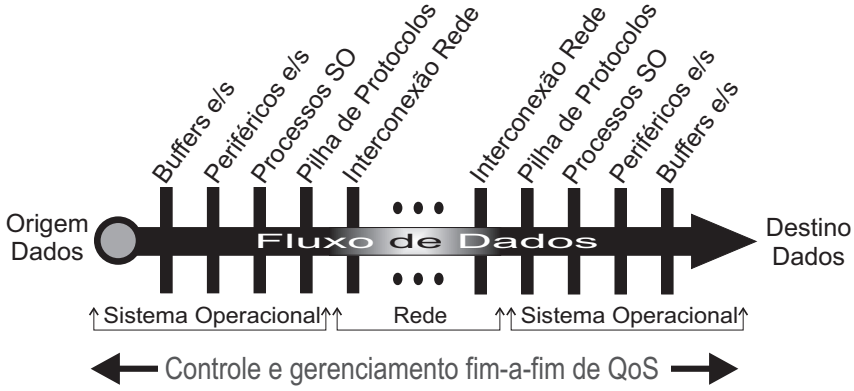


Figura 1: Exemplo de fluxo fim-a-fim.

Entretanto, internamente a dispositivos computacionais, i.e., não necessariamente equipamentos conectados a redes, também faz-se necessária a utilização de QoS para atender requisi-

tos próprios de cada aplicação. Por exemplo, a tarefa de exibição de vídeo exige determinada taxa de amostragem de quadros por segundo. Decodificação de vídeo, decodificação de áudio e dimensionamento de imagem fazem parte dessa tarefa. Se essas tarefas devem ser executadas com um mínimo de latência ou atraso, os contratos de QoS podem garantir o atendimento dos requisitos.

Da mesma forma, sistemas operacionais de propósito geral, que cada vez mais lidam com ambientes multitarefa e com recursos multimídia, dependem do correto gerenciamento de recursos para que os resultados finais sejam os melhores possíveis. Por exemplo, durante a gravação de um CD-R, a manipulação de arquivos dentro do disco rígido não pode interromper o fluxo de dados que é reservado ao processo de gravação. Se tal gerência de recursos não existe, corre-se o risco de perder a mídia virgem se o SO for utilizado simultaneamente por outras aplicações.

Existe, também, uma gama de aplicações onde o uso de QoS auxilia no cumprimento de requisitos, que são sistemas embarcados de tempo-real, i.e., sistemas dedicados que desempenham uma função específica com restrições de tempo. Estes sistemas podem estar presentes em diversos tipos de aplicações. Exemplos dessas aplicações são redes de comunicação (telefonia sem fio), tele-medicina (cirurgia remota), automação de processos de fabricação (transformação de metais), e sistemas de defesa (sistemas para missões em aviação) [2].

Sistemas de tempo-real podem ser classificados em dois tipos: *soft* e *hard* [3]. Aplicações *soft real-time* são aquelas nas quais existem restrições de tempo, mas atrasos no cumprimento de requisitos temporais são tolerados. Como exemplos dessa classe de aplicações pode-se citar teleconferência, comunicação digital de voz e exibição de vídeo. Aplicações *hard real-time*, de semelhante modo, possuem restrições de tempo mas atrasos não são tolerados, i.e., a resposta correta atrasada torna-se a resposta errada. Exemplos são o sistema controle interno do motor de um automóvel e marca-passo. Neste último exemplo, pode-se verificar que o cumprimento de restrições de tempo é vital.

Dentro do contexto de aplicações de sistemas embarcados, MPSoCs (do inglês, *Multiprocessor Systems on Chip*) representam uma tendência onde conceitos utilizados em sistemas distribuídos podem ser aplicados. Redes intra-chip são usadas como meio de intercomunicação entre os diversos elementos de processamento que integram o chip. Assim, com diversos nodos em uma rede intra-chip, a aplicação de conceitos de Qualidade de Serviço é desejável e representa a viabilização de aplicações que, sem a presença de mecanismos de controle de qualidade, não seriam possíveis.

Escalonadores de processos em sistemas de tempo-real são os responsáveis por delegar a ordem de execução de processos em um processador. A adição de QoS em escalonadores pode auxiliar desenvolvedores de aplicações de tempo-real a atingirem os requisitos temporais intrínsecos às aplicações. Através deste trabalho, pretende-se encontrar uma solução para provimento de QoS no escalonamento de processos de sistemas embarcados de tempo-real, buscando verificar a aplicabilidade em diferentes áreas dentro de sistemas embarcados.

1.1 Motivação

Previsibilidade e flexibilidade são características necessárias a determinadas aplicações embarcadas que possuem requisitos de tempo. Para obter essas características faz-se o uso de QoS. Sistemas operacionais embarcados (SOE) de tempo-real devem fornecer mecanismos de configurabilidade para o atendimento dos requisitos de QoS. Assim, o desenvolvedor de aplicações embarcadas tem a possibilidade de parametrizar o comportamento de suas aplicações.

Parte do atendimento à qualidade está sob responsabilidade do escalonador de processos de um SOE. Dentre os trabalhos relacionados estudados, não encontrou-se suporte para reserva de processamento para tarefas *hard real-time*. Assim, deseja-se implementar um algoritmo de tempo real com reservas, modificando-o para suprir a necessidade de tarefas *hard real-time*.

1.2 Objetivo

O objetivo desse trabalho consiste em modelar e implementar um sistema para provisão de QoS para sistemas operacionais de tempo-real. Esse sistema irá focar na provisão de QoS para o escalonador do sistema operacional. Será utilizado e adaptado o SO embarcado EPOS, para prover QoS às aplicações. Como método de validação são utilizados conjuntos de *benchmarks* através de distribuições matemáticas.

1.3 Organização do Documento

Este documento está organizado da seguinte forma: no Capítulo 2 encontram-se os principais conceitos relacionados à provisão de QoS juntamente com os trabalhos relacionados demonstrando o estado-da-arte em QoS para sistemas operacionais. A seguir, no Capítulo 3 são apresentados a arquitetura de *software* proposta, o algoritmo de escalonamento com reservas proposto e as ferramentas de auxílio para execução de testes e análise de resultados. Após, no Capítulo 4 são apresentados testes comparativos entre os algoritmos apresentados. Por fim, no Capítulo 5 são apresentadas as conclusões e propostas para trabalhos futuros.

2 Referencial Teórico

Neste capítulo são apresentados o referencial teórico e os trabalhos relacionados com este trabalho. No referencial teórico, são apresentados conceitos sobre sistemas de tempo-real, sistemas embarcados e provisão de QoS para sistemas operacionais. Na seção de trabalhos relacionados, são apresentados os principais trabalhos que propõe soluções para a provisão de QoS em sistemas operacionais.

2.1 Sistemas Embarcados

Sistemas embarcados são aqueles projetados para um fim específico, i.e., são sistemas dedicados. Exemplos destes são aparelhos de GPS, informatização dentro de um automóvel, câmeras fotográficas digitais. Geralmente, esses sistemas são formados por um ou mais microprocessadores, memória e meio de interconexão entre os elementos de sistema. Cada processador de um sistema embarcado executa um sistema operacional e, em sistemas multiprocessados, pode executar apenas parte do SO.

Cada vez mais são encontrados equipamentos com capacidade de fornecer informação, entretenimento e comunicação que, por sua natureza complexa, exigem mecanismos para atingir tipos de requisitos como, por exemplo, performance, consumo de energia e tempo de resposta. Como exemplo de equipamentos podem ser citados os de telefonia móvel, que a cada geração incorporam funções que vão além da funcionalidade básica de um telefone. Vídeo, música e fotografia digital fazem parte das funcionalidades de modernos equipamentos celulares que, apesar do número de funcionalidades, tendem a ser continuamente decrescentes em tamanho.

Devido a crescente demanda de mercado por produtos novos, contendo funcionalidades e restrições complexas, métodos para o desenvolvimento acelerado de aplicações são necessários. *Frameworks*, *SoCs*, *MPSoCs* e QoS são exemplos de métodos que permitem aos desenvolvedores abstração, configurabilidade e reuso durante o processo de criação.

2.1.1 SoC

O constante avanço tecnológico tem possibilitado a integração de múltiplos módulos de *hardware*, como processadores, memória e periféricos para realização de funções específicas, no mesmo circuito integrado (CI). Um CI com múltiplos módulos é denominado *System-on-Chip* (SoC, do inglês, sistema em um chip) [4] [5]. Essa possibilidade de acréscimo de módulos

tem resultado no desenvolvimento de equipamentos com maior desempenho e maior número de funcionalidades.

A complexidade dos SoCs cresce juntamente com o número de módulos agregados em um mesmo CI e a pressão de *time-to-market* exige que o processo de desenvolvimento de tais arquiteturas seja, além de eficaz, eficiente. Para atender esses requisitos faz-se necessária a aplicação de técnicas de re-uso [6], que prega a padronização de interfaces e a modularização de diferentes componentes de modo que haja pouca interdependência entre os mesmos.

A performance é outro quesito importante quando trata-se com SoCs. Para determinadas aplicações, o uso de somente um processador para a execução de todo o sistema é uma alternativa mais custosa em termos de performance e/ou consumo de energia comparado a sistemas compostos por mais de um processador [7].

2.1.2 MPSoC

Um SoC multiprocessado é chamado de MPSoC (do inglês, Multiprocessor System on Chip) [8]. As arquiteturas de MPSoC lembram as consagradas arquiteturas de multi-processadores, mas MPSoC adiciona custo e consumo de energia às preocupações de projeto de sistemas multiprocessados abrindo, assim, um leque de opções de estudo [9]. Na figura 2 é mostrado um exemplo básico um MPSoC. Nesse exemplo três elementos de processamento (EP), memória e interface de entrada e saída se comunicam através de um barramento.

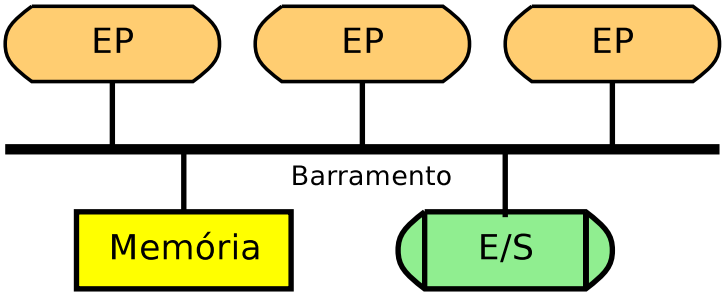


Figura 2: Exemplo de arquitetura de um MPSoC.

Aplicações de sistemas embarcados tipicamente requerem concorrência real, e não somente uma simulação através de escalonamento de tarefas por divisão de tempo. Por exemplo, a codificação de vídeo no formato MPEG-2 requer diversas tarefas executando ao mesmo tempo, onde uma depende da saída da outra. Com o vídeo entrando no sistema a 30 quadros por segundo, as tarefas devem executar em paralelo para atender as restrições de tempo. O típico MPSoC é um sistema multiprocessado heterogêneo, isto é, possui EPs de diferentes tipos. A memória desses sistemas pode ser distribuída pela arquitetura também de forma heterogênea, assim como o meio de interconexão dos diversos EPs [9].

Neste contexto, projetistas estão se deparando com novos desafios dada a complexidade dos futuros MPSoCs e do grande espaço de projeto que apresenta várias alternativas que precisam

ser exploradas durante a definição da arquitetura do sistema [10], tais como: (*i*) estrutura de interconexão, (*ii*) arquitetura de software, (*iii*) desempenho, (*iv*) consumo de energia e (*v*) tempo de projeto (*time-to-market*).

Em (*iii*), (*iv*) e (*v*), é onde QoS pode contribuir para o atendimento dos requisitos das aplicações. Na parte de desempenho, QoS pode contribuir na garantia de largura de banda e tempo de processamento. Conforme proposto em [11], o consumo de energia (*iv*) pode ser considerado no gerenciamento de QoS do sistema. Em (*v*), quando restrições de performance precisam ser testadas e ajustadas manualmente pelo projetista, QoS pode ser a solução para diminuir o tempo de projeto automatizando esse processo.

2.2 QoS para Sistemas Operacionais

QoS para sistemas operacionais trata, então, da qualidade de fluxos de dados de modo fim-a-fim. Dentro de um sistema maior, como, por exemplo, um sistema distribuído, o sistema operacional executa uma parte fundamental para manter a qualidade requerida por uma aplicação. Cada parte que compõe um fluxo fim-a-fim é importante e deve ter seus parâmetros de qualidade gerenciáveis. A comunicação de rede também faz parte do fluxo fim-a-fim e, por sua vez, possui características distintas, as quais também devem ser gerenciáveis. Parte do controle do subsistema de rede reside no sistema operacional, como, por exemplo, a pilha de protocolos. A seguir, serão apresentados os principais conceitos relativos ao provimento de QoS.

2.2.1 Mecanismos de Provisão de QoS

Provisão de QoS é formada pelos seguintes componentes:

- **Mapeamento**: executa a função de tradução automática entre representações de QoS em diferentes níveis de sistema, i.e., sistema operacional, camada de transporte, rede, etc. Isso permite o usuário abstrair níveis inferiores de especificação de requisitos. Por exemplo, a especificação de QoS do nível de transporte da rede deve expressar requisitos de fluxos em termos de nível de serviço, média e pico da largura de banda, variação de atraso, *jitter*, limites de perda e atraso. Para teste de admissão e alocação de recursos, esta representação deve ser traduzida em algo mais significativo para o sistema. Por exemplo, uma contrato de serviço de QoS de alta disponibilidade pode ser traduzido em requisitos específicos de largura de banda e atraso, e.g., reserva-se 50% da banda utilizando um determinado algoritmo de escalonamento de pacotes que prioriza o menor atraso possível.

- **Teste de Admissão**: é responsável pela comparação dos requisitos de recursos originados do pedido de qualidade contra os recursos disponíveis no sistema. A decisão se um novo pedido pode ser acomodado depende geralmente das políticas de gerência de recursos do

sistema e da disponibilidade de recursos. Uma vez que o teste de admissão teve sucesso em um módulo particular de recurso, recursos locais são reservados.

- **Protocolos de Reserva de Recursos**: organiza a alocação de recursos do sistema e rede de acordo com a especificação do usuário. Por exemplo, para uma reserva fim-a-fim que passa por diversos nodos da rede, primeiramente, o protocolo de reserva de recursos interage com o roteamento baseado em QoS para estabelecer um caminho através da rede. Então o mapeamento de QoS e controle de admissão em cada recurso visitado (e.g. CPU, memória, entrada/saída, roteadores, etc.) alocam recursos fim-a-fim. O resultado final é que os mecanismos de controle de QoS, tais como escalonadores de pacotes em nível de rede e e escalonadores de processos são corretamente configurados.

2.2.2 Mecanismos de Gerência de QoS

Para manter os níveis acordados de QoS, é comum que a reserva de recursos não seja suficiente. Assim, o gerenciamento de QoS é freqüentemente requerido para garantir que a qualidade contratada é sustentável. Gerência de QoS de fluxos é funcionalmente semelhante ao controle de QoS. Entretanto, ele opera em uma escala de tempo mais lenta, isto é, sobre longos intervalos de monitoramento e controle. Os mecanismos de gerência de QoS fundamentais incluem:

- **Monitoramento**: permite que cada nível do sistema observe as ações dos níveis de QoS alcançados pelas camadas inferiores. A parte de monitoramento muitas vezes representa uma parte fundamental no ciclo de realimentação do sistema, que irá utilizar as informações capturadas para atuar sobre o nível da qualidade. Algoritmos de monitoramento operam sobre diferentes escalas de tempo. Por exemplo, eles podem executar como parte do escalonador para medir a performance individual dos fluxos correntes. Neste caso, as estatísticas coletadas podem ser usadas para controlar o escalonamento de pacotes e para controle de admissão.

- **Manutenção**: compara a qualidade monitorada com a esperada e realiza operações de ajuste nos recursos a fim de sustentar a qualidade entregue.

- **Degradação**: emite uma indicação de QoS ao usuário quando é determinado que as camadas mais baixas falharam na manutenção da qualidade do fluxo e nada pode ser feito pelo mecanismo de manutenção. Em resposta a tal indicação, o usuário pode escolher ou se adaptar ao nível disponível de qualidade ou escalar a um nível reduzido de QoS, i.e., realizar uma renegociação fim-a-fim.

- **Disponibilidade**: permite à aplicação especificar o intervalo sobre um ou mais parâmetros (atraso, *jitter*, largura de banda, perda e sincronização) que podem ser monitorados e a aplicação informada do desempenho entregue através de um sinal de QoS.

- **Escalabilidade**: compreende filtragem de QoS (manipula os fluxos enquanto eles progridem através do sistema de comunicação) e mecanismos de adaptação de QoS. Muitas aplicações de mídia contínua exibem robustez na adaptação à flutuações em QoS fim-a-fim. Baseado na política de gerenciamento fornecida pelo usuário, adaptação de QoS nos sistemas finais podem tomar ações para corrigir e escalar os fluxos apropriadamente.

2.3 Trabalhos Relacionados

Nesta seção serão mostrados trabalhos relacionados que utilizam QoS para oferecer garantias de serviços. Primeiramente será apresentado um SO de propósito geral modificado para atender requisitos básicos de QoS. A seguir, será mostrado um *framework* para provisão de QoS. Após, explana-se brevemente um trabalho que implementa decodificação de vídeo distribuída dentro de um MPSoC. Por fim, será descrito o algoritmo de escalonamento R-EDF, que fornece reservas de tempo de processamento para processos com requisitos de tempo-real.

2.3.1 Eclipse/BSD

Eclipse/BSD é um sistema operacional derivado do FreeBSD [12]. Voltado para aplicações de servidores, fornece um suporte de QoS flexível e adaptável. Os principais componentes elementares dessa arquitetura são: *(i)* uso de escalonadores hierárquicos de recursos, usando divisão proporcional, *(ii)* noção de *reserva* e sua implementação como um sistema de arquivos, *(iii)* mecanismo de marcação de aplicações para a associação com sua *reserva* e *(iv)* um esquema de controle de acesso e admissão, de onde surge a noção de *domínio de reserva*.

O principal objetivo do Eclipse/BSD é prover suporte de QoS para um grande conjunto de aplicações para servidores sem a necessidade de modificações significativa nessas aplicações. Por exemplo, se uma aplicação possui conexões com diversos clientes, não se deseja que exista uma instância da aplicação para cada cliente com o objetivo de se beneficiar das políticas diferenciadas de QoS. Adicionalmente, têm-se como objetivo o de fornecer uma plataforma de gerência de recursos com capacidade de implementar um grande conjunto de necessidades relativas a QoS.

Escalonadores

Para escalonadores de CPU, gerenciamento banda de disco e rede, faz-se o uso de escalonadores hierárquicos, de divisão proporcional. Os escalonadores de recurso são dinamicamente reconfiguráveis de maneira que a hierarquia de escalonamento e/ou pesos de divisão possam ser alterados sem a necessidade de parar o escalonador. Essas configurações são sujeitas aos controles de acesso e admissão.

Escalonadores hierárquicos, de divisão proporcional, suportam uma noção geral de *reservas*. Cada nodo da hierarquia representa uma *reserva* com sua *parte* na divisão igual ao seu peso

dividido pela soma de todos os filhos de seu pai na hierarquia, incluindo seu próprio peso. Na Figura 3 é mostrado como a divisão proporcional para largura de banda para acesso a disco. Os nodos A e B representam, cada, 50% da *reserva* sobre a largura de banda, pois ambos têm peso 1. No seguinte nível da hierarquia, os nodos C, D, E e F possuem 25% de largura de banda cada.

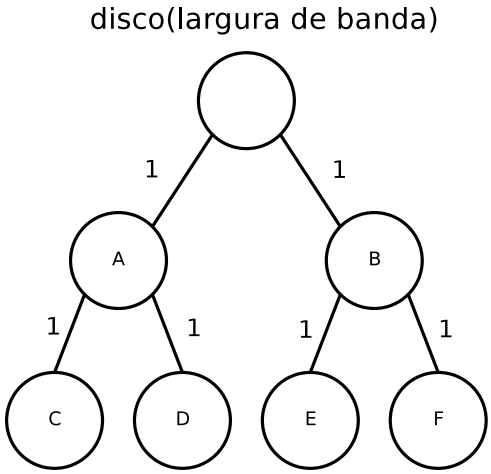


Figura 3: Hierarquia de escalonadores.

São utilizados dois tipos de nodos de reserva em uma hierarquia de escalonamento: escalonadores e filas de pedidos. Os nodos da hierarquia implementam um algoritmo de escalonamento para selecionar pedidos de reserva de recursos para os nodos filhos. Nodos de filas de pedidos são os pontos onde os recursos são inicialmente enfileirados. Esses nodos de filas são sempre folhas em uma hierarquia. Por exemplo, na Figura 3, o nodo C é uma fila onde pedidos de acesso a disco podem ser enfileirados.

Reservas

Foi desenvolvido um sistema de arquivos, que na hierarquia de arquivos UNIX, encontra-se em */reserv*. Este sistema de arquivos fornece uma API (do inglês, *Application Program Interface*) e uma hierarquia de nomes, através dos quais é possível acessar, usar e reconfigurar os escalonadores de recursos. Os diretórios contidos em */reserv* correspondem aos nodos da hierarquia de escalonadores e representam *reservas*. Cada diretório dentro de */reserv* é uma *reserva*. A API fornecida pelo sistema de arquivos em */reserv* fornece meios para adicionar e remover *reservas* e alterar os pesos dos escalonadores por toda hierarquia. Cada recurso é representado por uma *reserva* como, por exemplo, */reserv/cpu*. A estrutura de diretórios dentro de cada recurso representam as atuais *reservas* que o recurso atende. Por exemplo, */reserv/wd0/r1* e */reserv/wd0/r2* representam as reservas *r1* e *r2* da largura de banda do disco *wd0*.

Marcação

Um importante aspecto desse sistema é a associação de uma *reserva* com uma operação em um objeto. As operações em objetos incluem: leitura/escrita de arquivo, envio de mensagem pela rede e execução de um processo. As *reservas* correspondentes são, respectivamente, largura de banda de disco, largura de banda na transmissão através da interface de rede e ciclos de execução da CPU.

Para leitura/escrita em arquivo, *f*, o descritor de arquivo correspondente a esse arquivo é marcado com uma *reserva*. A *reserva* deve ser um diretório de fila referende ao dispositivo de armazenamento onde se encontra o arquivo *f*. Para uma *socket s*, a associação ocorre da mesma forma. Mas existe o problema de marcar *sockets* que ainda não foram conectadas, pois dependem do endereço de destino para saberem a qual dispositivo de rede deverá ser associada. Assim, criou-se um mecanismo de marcação atrasada para esses casos. Também é fornecido meios de modificar a marcação do descritor de arquivo.

As marcas são utilizadas para determinar a fila correta para requisições de entrada/saída baseadas em descritores de arquivos marcados. Por exemplo, se *fd* é o descritor de arquivo marcado com o diretório de fila *q*, então todo fluxo de entrada/saída será enfileirado no nodo correspondente ao diretório de fila *q*. Assim, esse fluxo passa pelos algoritmos implementados na hierarquia de diretórios, dividindo, assim, a largura de banda de acesso ao dispositivo.

Domínios de Reserva

Controle de acesso e de admissão aplicados o sistema de arquivos em */reserv* oferecem uma oportunidade de definir a noção de *domínio de reserva*. Conceitualmente, controle de acesso e admissão são usados para garantir ou negar o direito a acesso, uso ou reconfiguração do sistema de arquivos */reserv*. Como */reserv* é o centro de toda gerência de recursos no sistema, o conjunto de permissões que um determinado processo tem nesse sistema de arquivos é denominado *domínio de reserva* do processo. As credenciais de um processo incluem o identificador do processo (PID) e os tradicionais identificadores de usuário (UID) e grupo (GID). O controle de acesso e admissão oferece restrições e direitos no uso de diretórios de filas para marcação, criação de novas *reservas* e *subreservas*, mudanças nos pesos de escalonamento e capacidades das filas, entre outros. A utilização dessa noção de *domínio de reserva* é mais aparente em aplicações cliente-servidor, onde faz-se necessário um controle preciso sobre os recursos.

2.3.2 QoSOS

QoSOS é uma arquitetura genérica (*framework*) para provisão de QoS em sistemas operacionais. O desenvolvimento dessa arquitetura seguiu-se à análise de algumas soluções apresentadas na literatura e à percepção de semelhanças funcionais entre elas. A arquitetura QoSOS permite reutilizar funções comuns e definir uma organização interna que seja equivalente nos diferentes

sistemas, facilitando a definição de mecanismos de orquestração dos recursos do sistema geral como um todo.

A arquitetura QoSOS foi definida a partir da especialização e extensão dos *frameworks* para provisão de QoS em ambientes genéricos de processamento e comunicação, conforme descrito em [13]. Esses *frameworks* identificam conjuntos de funções recorrentes de provisão de QoS em vários subsistemas, como redes de comunicação, sistemas operacionais e plataformas distribuídas. A estruturação sob a forma de *frameworks* visou facilitar a identificação dos pontos de flexibilização (*hot-spots*) que devem ser preenchidos para descrever a funcionalidade de um ambiente específico.

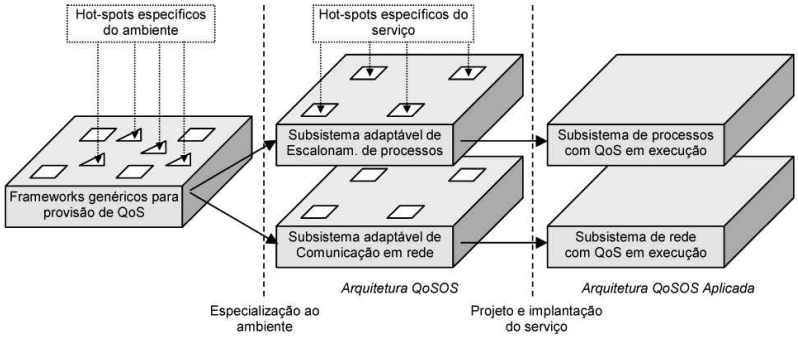


Figura 4: *Hot-spots* da arquitetura modelada a partir dos *frameworks* genéricos.

A Figura 4 mostra como os tipos de *hot-spots* podem ser completados para a construção de uma arquitetura de provisão de QoS em sistemas operacionais. Os *frameworks* genéricos definem as estruturas comuns para a provisão de QoS nos vários subsistemas que participam do fornecimento do serviço fim-a-fim. A primeira etapa de especialização é feita para que sejam incluídas funcionalidades específicas de sistemas operacionais, como os mecanismos pertinentes aos subsistemas de escalonamento de processos e de comunicação em rede. Na etapa seguinte, são definidos os aspectos relacionados à provisão do serviço, como o conjunto de políticas de QoS que cada um dos subsistemas disponibilizará a seus usuários.

QoSOS, sendo uma arquitetura genérica, oferece flexibilidade e configurabilidade para o provimento de QoS em sistemas operacionais. Porém, tais qualidades implicam no aumento do tamanho do sistema, o que não é desejável para sistemas operacionais embarcados.

Frameworks para compartilhamento de recursos

Os *frameworks* para compartilhamento de recursos se baseiam no conceito de recurso virtual para modelar os mecanismos de alocação e escalonamento. Recursos virtuais são parcelas de utilização de um ou mais recursos reais distribuídas entre os fluxos submetidos pelos usuários. Para facilitar o emprego de vários algoritmos de escalonamento sobre um mesmo recurso e, assim, oferecer um conjunto amplo e flexível de serviços em um mesmo sistema, os recursos virtuais são dispostos em uma estrutura chamada de árvore de recursos virtuais. Cada recurso real possui uma árvore de recursos virtuais associada, embora uma mesma árvore de recursos

possa representar a estrutura de escalonamento sobre mais de um recurso real. Um exemplo de árvore sobre vários recursos é o escalonamento de processos em sistemas multiprocessados.

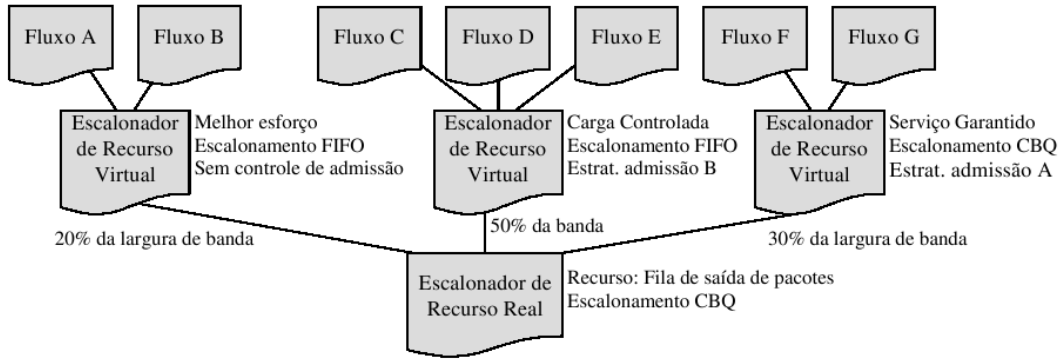


Figura 5: Exemplo de árvore de recursos virtuais para um canal de comunicação, na provisão de serviços integrados.

Em uma árvore de recursos virtuais, as folhas representam os recursos virtuais. A raiz da árvore corresponde ao escalonador de mais baixo nível da hierarquia, aquele que realmente distribui o tempo de uso do recurso real entre os nodos filhos. Adicionalmente, este escalonador, denominado escalonador de recurso raiz, pode permitir que os recursos virtuais filhos utilizem diferentes recursos reais de um mesmo tipo. Os nodos intermediários da árvore são recursos virtuais especializados, responsáveis por ceder a sua parcela de utilização do recurso real aos seus recursos virtuais filhos. Denominados escalonadores de recursos virtuais, esses nodos podem ter acesso a mais de um recurso real por vez, para também permitir que seus nós filhos sejam atendidos simultaneamente. A Figura 5 ilustra um exemplo de árvore de recursos virtuais.

Cada escalonador de recurso virtual está associado a uma categoria de serviço e às políticas de provisão de QoS correspondentes, como as estratégias de escalonamento e de admissão, além de um componente de criação de recursos virtuais. Para efetivar a criação de um recurso virtual, esse componente executa tarefas como a adição do recurso virtual à lista de responsabilidades do escalonador e a configuração dos módulos de classificação e de policiamento.

Frameworks para orquestração de recursos

Na área de atuação dos sistemas operacionais, vários são os recursos que devem ter seus mecanismos de escalonamento e de alocação gerenciados de forma integrada, em um processo de orquestração dos recursos de todo o ambiente. Na arquitetura QoSOS, a modelagem da orquestração de recursos é apresentada pela especialização de dois *frameworks* distintos: o *framework* para negociação de QoS e o *framework* para sintonização de QoS.

O *framework* para negociação de QoS modela os mecanismos de negociação e mapeamento que operam durante as fases de solicitação e estabelecimento de serviços, além dos mecanismos de admissão que atuam somente na fase de estabelecimento. Já o *framework* para sintonização de QoS modela os mecanismos de sintonização e monitoração que atuam na fase de manutenção

do serviço.

Ao receber uma requisição de serviço, o controlador de admissão do sistema operacional deve verificar a viabilidade de aceitação do serviço naquele nível de abstração. Para isso, ele repassa a requisição ao agente de orquestração do sistema operacional, responsável por identificar todos os recursos reais que podem estar envolvidos no fornecimento do serviço e, então, distribuir entre eles as parcelas de responsabilidade sobre a provisão da QoS especificada. No caso de uma aplicação distribuída, o orquestrador do sistema operacional identificará que CPUs e *buffers* de comunicação são recursos que devem participar do oferecimento do serviço. A negociação de QoS em um sistema operacional é feita de forma centralizada, já que um único agente pode ter o conhecimento sobre todos os recursos do ambiente.

As aplicações multimídia distribuídas devem ter garantidas as suas necessidades sobre cada uma das *threads* que compõem seus processos, bem como sobre as *threads* que executam a pilha de protocolos. Além disso, os *buffers* de comunicação compartilhados devem ser capazes de encaminhar os pacotes de acordo com a parcela de QoS atribuída à estação pelo protocolo de negociação de rede e, por isso, a forma de implementação do subsistema de rede deve ser considerada na distribuição das responsabilidades. As estratégias de negociação definem como será a política de orquestração, baseando-se na implementação de subsistemas específicos.

A partir das parcelas de responsabilidade atribuídas a cada recurso, são acionados os mecanismos de mapeamento, consistindo na tradução da categoria de serviço (e dos parâmetros associados), especificada na solicitação do serviço, para as categorias de serviço (e parâmetros associados) relacionadas diretamente com a capacidade de operação de cada recurso real envolvido. O mecanismo de controle de admissão associado a cada recurso deve, então, ser acionado, a fim de verificar a viabilidade de aceitação do novo fluxo, utilizando-se das estratégias de admissão de recursos virtuais em cada um dos escalonadores escolhidos. Se todos os controladores de admissão responderem de forma afirmativa, os mecanismos de criação de recursos virtuais são acionados. Caso contrário, a requisição pode ser imediatamente negada, ou a negociação pode ser reiniciada, redistribuindo-se as parcelas de responsabilidade.

Durante a fase de manutenção de um contrato de serviço, ajustes sobre o sistema podem ser necessários, para que sejam asseguradas as especificações de QoS já requisitadas. A monitoração dos recursos reais visa a identificação de disfunções operacionais, seja por parte do usuário (e.g. fluxos submetidos fora da caracterização do tráfego), seja por parte do sistema (e.g. falha nos recursos, erros no cálculo das reservas). Os monitores devem emitir alertas ao mecanismo de sintonização na presença de algum distúrbio. As ações de sintonização podem envolver desde pequenos ajustes de parâmetros em determinados escalonadores, até a solicitação de uma renegociação geral da QoS.

Framework para adaptação de serviços

Embora os *frameworks* genéricos para provisão de QoS ofereçam a projetistas o conceito de pontos de flexibilização (*hot-spots*) específicos de serviço, permitindo a modelagem de sistemas

adaptáveis, esses pontos apresentam relações indiretas de dependência entre si que dificultam a manutenção da consistência do sistema face a adaptações. Nesse contexto, a implementação de "meta-mecanismos", que automatizem a adaptação do sistema a novos serviços ou a novas políticas de provisão de QoS, e que observem questões como manutenção de consistência e restrições de reconfiguração relacionadas a segurança, é altamente desejável. O *framework* para adaptação de serviços foi elaborado neste trabalho para preencher parte dessa lacuna deixada pelos *frameworks* genéricos para provisão de QoS, tendo, no entanto, uma abordagem específica para sistemas operacionais.

As ações de adaptação requeridas pelos administradores do sistema, ou por um mecanismo externo, devem ser controladas por um gerente de adaptação, responsável por receber as requisições, fazer verificações sobre a possibilidade de aceitação, inserir ou substituir o componente alvo e, finalmente, atualizar as referências nos mecanismos a ele relacionados. Para a criação de um serviço inteiramente novo, todos os componentes que implementam as políticas de provisão e QoS devem ser fornecidos ao gerente, juntamente com a localização da nova categoria na hierarquia de categorias de serviço.

Parte dos testes a que se refere o parágrafo anterior compreende a verificação de segurança da inserção do componente, que é delegada pelo gerente de adaptação a um agente específico. De um modo geral, o agente de verificação de segurança deve analisar cada novo componente levando em conta os seguintes aspectos básicos:

- Confiabilidade. O fornecedor do componente deve ser confiável.
- Restrição de contexto. As ações descritas pelo componente devem estar restritas ao contexto no qual o componente será aplicado.
- Isolamento. As ações descritas pelo componente, se logicamente erradas, não podem prejudicar a provisão de serviços para outras categorias ou a operação de outros subsistemas.

Se as verificações foram bem sucedidas, o gerente de adaptação submete a implementação do componente à porta de adaptação a ela correspondente. Portas de adaptação são as estruturas existentes no sistema operacional responsáveis por disponibilizar a implementação do componente aos mecanismos que a utilizarão. Como um exemplo de porta de adaptação, pode-se citar o subsistema de módulos de *kernel* do Linux. Finalmente, o gerente atualiza as estruturas que devem fazer referência ao novo componente, como um escalonador faz a um componente de criação ou a uma estratégia de escalonamento.

Um outro detalhe importante a ser observado está na remoção de componentes do sistema. Além da verificação de segurança, que confirma se o solicitante está autorizado para a ação, um outro teste, chamado de verificação de consistência, deve ser executado. O teste de consistência da remoção tem a responsabilidade de verificar se a remoção de um componente não acarretará o mau funcionamento ou total parada de outros componentes. Uma estrutura de dependências deve, então, ser mantida.

Nota-se que a funcionalidade dos mecanismos de adaptação pode ser aplicada não somente à infra-estrutura de provisão de qualidade de serviço, como também para outras partes do sistema operacional, como o subsistema de rede (pilha de protocolos), o gerenciamento de *drivers*, o sistema de arquivos, entre muitos outros. Obviamente, essa capacidade deve ser provida pelo kernel, atribuindo a esses subsistemas o suporte a portas de adaptação.

2.3.3 MPEG4 em MPSoC com Rede Intra-chip

Pastrnak [14] explora o conceito de QoS para sistemas embarcados MPSoC que utilizam redes intra-chip como estrutura de interconexão. Neste contexto, um MPSoC pode ser visto como uma micro-rede de componentes, onde a rede é o meio responsável pela comunicação entre os elementos de processamento do sistema [15]. Toda rede intra-chip possui um protocolo de comunicação que determina como os núcleos do MPSoC são conectados à rede, bem como a maneira como os dados trafegam da origem ao destino [16].

Com ênfase em aplicações de decodificação de vídeo digital, no trabalho é proposto um método de gerenciamento de QoS. Este utiliza fórmulas algébricas para computar a utilização de recursos, aplicando os parâmetros de qualidade previamente estabelecidos como, por exemplo, o número de triângulos gráficos a serem processados.

A classe de sistemas que dependem na predição dos tempos de execução da aplicação durante sua execução levando em conta a interdependência de dados é o enfoque principal desse trabalho. Como estudo de caso e prova de conceito, utiliza-se uma aplicação multimídia de decodificação de textura para objetos individuais em MPEG-4. Esta, por sua vez, possui tempos de execução de alta variação e, também, pode ter várias instâncias sendo executadas em paralelo. Cada instância é composta internamente de várias tarefas. Nessa aplicação é possível que um conjunto de objetos seja decodificado em paralelo, onde cada objeto tem suas características e comportamento. O mapeamento eficiente de tal aplicação implica em um problema de gerência, requerendo, assim, o uso de controle de QoS sobre os recursos da plataforma.

O modelo hierárquico de QoS proposto distingue os detalhes da aplicação e a interação entre diferentes instâncias da aplicação, que processam diferentes objetos. Também são considerados outros processos concorrendo com os recursos da aplicação de decodificação. A qualidade de serviço desejada alcança uma instância individual da aplicação.

Conforme demonstrado na Figura 6, a arquitetura é composta por dois gerenciadores que negociam entre si através de um protocolo de negociação. O *gerenciador global* (QoS Global) controla a performance total do sistema, enquanto que o *gerenciador local* (QoS Local) controla uma aplicação associada a determinados recursos alocados.

Cada aplicação é dividida em tarefas, que, por sua vez são divididas entre diferentes processos, e.g., leitura de cabeçalhos, quantização inversa. Para executar uma tarefa em um sistema multiprocessado, os processos são mapeados para processadores e recursos.

O modelo cliente-servidor usado para este trabalho possibilita o desacoplamento necessário

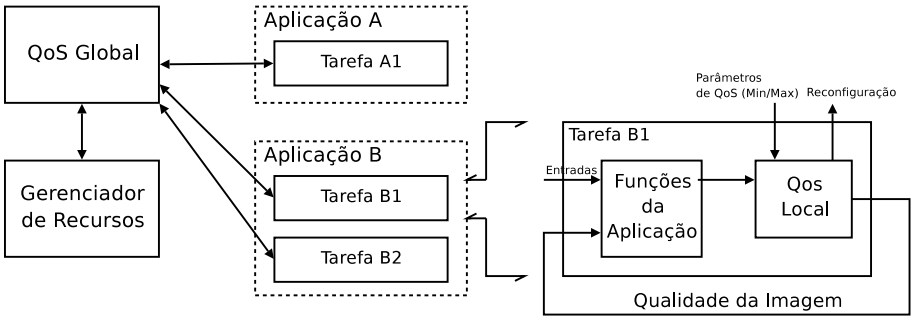


Figura 6: Hierarquia de QoS.

para a escalabilidade de um sistema MPSoC. Um sistema de gerência é utilizado para a otimização da utilização dos recursos. Entretanto, não são abordadas aplicações de tempo-real, motivando, assim, o estudo de uma arquitetura cliente-servidor para sistema com restrições de tempo.

2.3.4 R-EDF

Reservation-based Earliest Deadline First (R-EDF) [17] é um algoritmo de escalonamento preemptivo baseado em reservas. Esse algoritmo é baseado no conhecido *Earliest Deadline First* (EDF), porém implementa controle de admissão, suporte para aplicações de melhor esforço (*best-effort*) e degradação previsível em ambientes sobrecarregados.

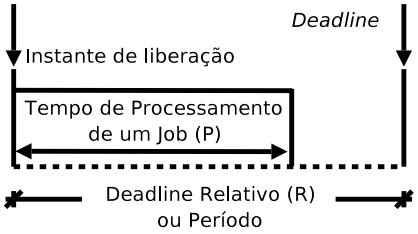
Para entender o funcionamento desse algoritmo, faz-se a distinção entre dois termos: *job* e *task*, do inglês, respectivamente, trabalho e tarefa. Uma *task* é um conjunto de *jobs*. Cada *job* é um conjunto de instruções de processamento que possui um *deadline*. Os *jobs* podem ser interdependentes entre si. Por exemplo, em uma decodificação de vídeo, decodificação é a *task*, que compreende a decodificação de diversos quadros, sendo cada um deles um *job*.

Na Figura 7(a) é ilustrado o modelo de um *job*. Denomina-se instante de liberação o momento que o *job* é disponibilizado para execução. A utilização de um *job* é representada pela equação $\theta(J) = \frac{P}{R}$, onde P é o tempo de processamento e R o *deadline relativo*, também denominado período. Na Figura 7(b) é ilustrado o modelo de uma *task* (tarefa), onde $T = J_1, J_2, ..., J_n$ para $n \geqslant 1$. A utilização de uma tarefa T é calculada efetuando a média das utilizações dos *jobs*: $\theta(T) = \frac{\sum_{i=1}^{n} \theta(J_i)}{n}$.

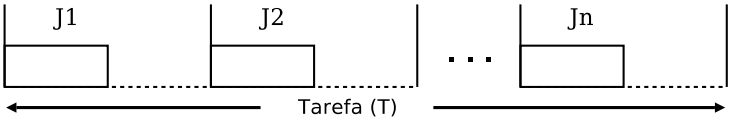
R-EDF suporta quatro classes de *tasks* de tempo-real e uma classe para tarefas sem restrições de tempo (melhor-esforço):

Periódicas de tempo constante (PTC): *jobs* de uma PTC possuem tempo de processamento e período constantes, como ilustrado na Figura 8(a).

Eventos: são um tipo especial de PTC que contém apenas um *job*, como ilustrado na Figura 8(b).



(a) Um *Job*



(b) Uma tarefa com n *jobs* dependentes

Figura 7: Modelos de tarefa e *job*.

Periódicas de tempo variável (PTV): *jobs* de uma PTV possuem período constante, porém têm tempo de processamento variável. Define-se um superperíodo Π como sendo um comportamento de processamento que se repete a cada w *jobs*. Na Figura 8(c) é ilustrada uma tarefa PTV exibindo o primeiro e o último superperíodos Π, cada um contendo w *jobs*.

Aperiódicas de utilização constante (AUC): *jobs* dessa classe possuem períodos arbitrários, i.e., podem variar a cada *job*. Também possuem tempo de processamento de larga variação, conforme ilustrado na Figura 8(d). Como geralmente não existem algoritmos para garantir *deadlines* de tarefas aperiódicas, essa classe está limitada a *jobs* de utilização constante, i.e., a utilização de toda a tarefa é previamente conhecida e não varia.

Melhor-esforço: tarefas sem requisitos temporais que não devem sofrer *starvation*.

Define-se utilização (θ) e utilização pico (ψ) como, respectivamente, a média de utilização de todos *jobs* da *task* e a utilização do *job* de maior tempo de processamento. Os parâmetros de entrada para a reserva são: classe, período, *deadline* relativo, utilização e utilização pico. Diferentemente das outras classes, a classe AUC informa ao escalonador um *deadline* novo a cada *job* gerado. Dentro deste modelo, cada *task* efetua a reserva de processamento para todos seus *jobs*. Dessa forma, uma *task* T faz requisição de $\theta(T)$ tempo de processamento. Por exemplo, se a tarefa possui 20% de utilização ($\theta(T) = \frac{1}{5}$), uma reserva de 20% será realizada para todos seus *jobs*. Caso um *job* ultrapasse seu tempo reservado, ele é colocado em estado *overrun*, conforme será explanado na Subseção 2.3.4.

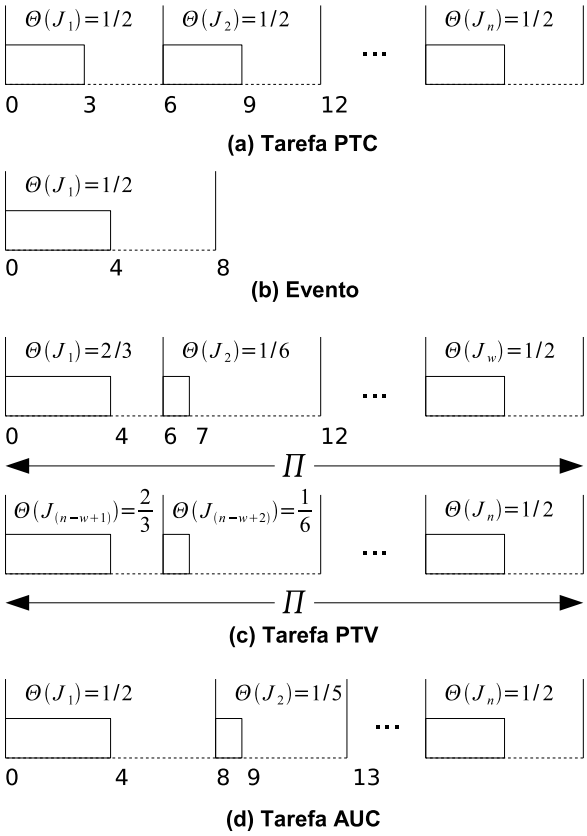


Figura 8: Múltiplas classes de tarefas.

Definições

A capacidade de um processador é definida como sendo 1, i.e., 100%. Um sistema com M processadores tem capacidade M. A capacidade de um processador é estatisticamente multiplexada entre tarefas de tempo-real e tarefas de melhor-esforço. A capacidade de *time-sharing* C_{TS} é a parte não reservada do tempo de processamento que é dividida entre todas aplicações de melhor-esforço. C_{TS} tem um limite inferior β ($C_{TS} \geqslant \beta$) para proteger tarefas de melhor-esforço. A capacidade de tempo-real C_{RTp} e utilização pico PC_{RTp} de um processador p ($1 \leqslant p \leqslant M$) são definidas como a soma da utilização e a utilização pico, respectivamente, de todas tarefas de tempo-real atribuídas ao processador. Assim, $C_{RTp} = \sum_{i=1}^{m} \theta(T_i)$ e $PC_{RTp} = \sum_{i=1}^{m} \psi(T_i)$, onde T_i ($1 \leqslant i \leqslant m$) são tarefas de tempo-real atribuídas ao processador p. O sistema é classificado como sobrecarregado se (1) $PC_{RTp} > 1$ para qualquer processador p, ou (2) $\sum_{p=1}^{M} PC_{RTp} > M - \beta$ para o todo o sistema. Caso contrário, o sistema é classificado como sobcarregado.

Na Figura 9 são ilustradas duas tarefas com três *jobs* cada. A tarefa A, da classe PTC, possui utilização $\theta(A) = 1/2$ e utilização pico $\psi(A) = 1/2$. A tarefa B, da classe PTV, possui utilização $\theta(B) = 1/3$ e utilização pico $\psi(B) = 1/2$. Se essas duas tarefas estiverem executando em um único processador, temos: $C_{RT} = \frac{1}{2} + \frac{1}{3} = \frac{5}{6} \cong 0.8333$ e $PC_{RT} = \frac{1}{2} + \frac{1}{2} = 1$. Se $\beta = 1/10$, i.e., 10% do processamento está reservado para tarefas de melhor esforço, então

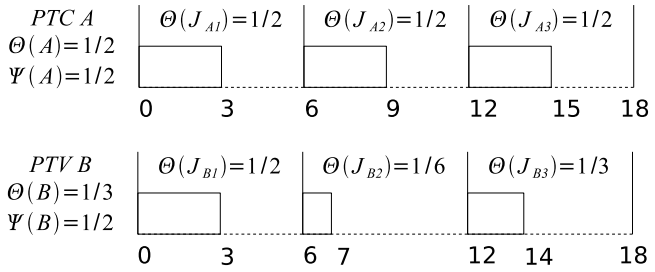


Figura 9: Exemplo de tarefas com respectivas taxas de utilização.

este sistema está sobrecarregado, pois

$$\sum_{j=1}^{M} PC_{RTj} > M - \beta$$

$$1 > 1 - 0.1$$

$$1 > 0,9$$

para $M = 1$. Com o sistema estando sobrecarregado, alguma tarefa poderá entrar no estado *overrun*, perdendo seu *deadline* (veja em 2.3.4).

Controle de Admissão

O controle de admissão analisa requisições de reserva de cada *task*. O algoritmo de controle de admissão, mostrado a seguir, verifica se os recursos disponíveis são suficientes para atender a requisição.

Passo 1 Inicialmente a capacidade de tempo-real C_{RTp} e capacidade pico PC_{RTp} de cada processador p são inicializadas com 0 $(1 \leqslant p \leqslant M)$, e a capacidade *time-sharing* C_{TS} é inicializada com M.

Passo 2 Uma tarefa de tempo-real com utilização θ e utilização pico ψ efetua requisição de reserva:

Se a capacidade *time-sharing* pode ser reduzida para admitir essa tarefa $C_{TS} - \theta \geqslant \beta$, e um processador p $(1 \leqslant p \leqslant M)$ pode atingir o requerimento $C_{RTp} + \theta \leqslant 1$

Então a tarefa é admitida e atribuída ao processador p, com: $C_{RTp} = C_{RTp} + \theta$; $PC_{RTp} = PC_{RTp} + \psi$; $C_{TS} = C_{TS} - \theta$

Senão a tarefa é rejeitada.

Passo 3 **Se** uma tarefa de tempo-real com utilização θ e utilização pico ψ, atribuída a um processador p, libera sua reserva, **então** $C_{RTp} = C_{RTp} - \theta$; $PC_{RTp} = PC_{RTp} - \psi$; $C_{TS} = C_{TS} + \theta$.

Escalonamento

Cada tarefa de tempo-real efetua uma reserva baseada em sua utilização θ. O escalonador, então, aloca θ% de tempo de processamento para todos os *jobs* daquela tarefa. Um *job* entra em estado *overrun* quando ele precisa de mais tempo de processamento que o tempo reservado. Causas comuns para uma tarefa entrar nesse estado são: (1) reserva insuficiente, (2) *overhead* de escalonamento ou temporização inexata e (3) estimativas erradas. Cada tarefa possui uma lista de *jobs* disponíveis (LJD). Quando um *job* é liberado pela tarefa para execução, esse *job* faz parte dessa lista LJD.

O algoritmo R-EDF seleciona uma tarefa de tempo real que esteja pronta baseada no *deadline* mais próximo de seu último *job*. Se não há tarefas de tempo-real disponíveis, o escalonador de tarefas de melhor-esforço é invocado. R-EDF protege o sistema quando este está sobrecarregado preemptando e colocando tarefas no estado *overrun* quando estas já utilizaram todo o tempo reservado. Isto ocorre mesmo que a tarefa possua o *deadline* mais próximo. Não existe a necessidade de proteger o sistema quando está em estado sobcarregado. Assim, quando o sistema está sobcarregado, R-EDF se comporta da mesma forma que o algoritmo EDF original. O Algoritmo R-EDF é descrito a seguir.

Passo 1 Selecione uma tarefa para execução:

1. **Se** alguma tarefa de tempo-real está pronta, **então** seleciona a tarefa cujo último *job* liberado para a lista LJD tenha o *deadline* mais próximo e execute seus *jobs*.
2. **senão** invoque o escalonador de tarefas de melhor-esforço.

Passo 2 O escalonador espera até a próxima unidade de tempo:

1. **Se** a tarefa que está executando termina a execução de seus *jobs*, i.e., a LJD está vazia, **então** ela entra no estado *esperando*.
2. **Senão se** o sistema está sobrecarregado e ainda há *jobs* na LJD da tarefa corrente e a tarefa utilizou seu tempo reservado, **então** ela entra o estado **overrun**.
3. Verificar todas tarefas pelo início de um novo período e colocá-las no estado **pronto**.

Passo 3 Vá para o passo 1.

Conforme ilustrado na Figura 10, uma tarefa de tempo-real possui três estados: (1) *esperando*: nenhum *job* disponível para execução; (2) *pronto*: há *jobs* prontos para execução; (3) *overrun*: há *jobs* prontos, mas a tarefa está indisponível para proteção do sistema.

Na Figura 11 é ilustrado o escalonamento de duas tarefas de tempo real em um sistema sobrecarregado, onde a tarefa B entra no estado *overrun*. A reserva para tarefas de melhor-esforço é $\beta = 1/6$. O *job* J_{B1} perde seu *deadline*, pois precisava de três unidades de tempo para executar, mas havia reserva para somente duas. Sua execução é continuada no segundo período. Apesar de entrar no estado *overrun*, a tarefa é executada por completo ao término do último período.

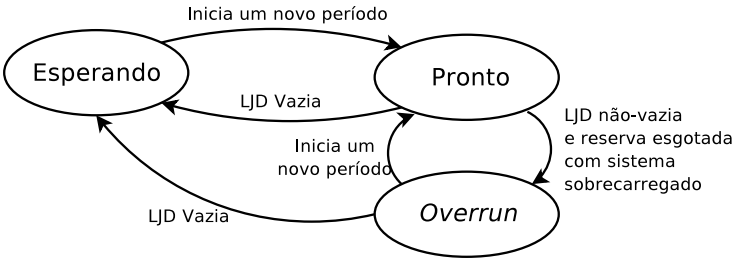


Figura 10: Diagrama de autômatos finitos de tarefas de tempo-real.

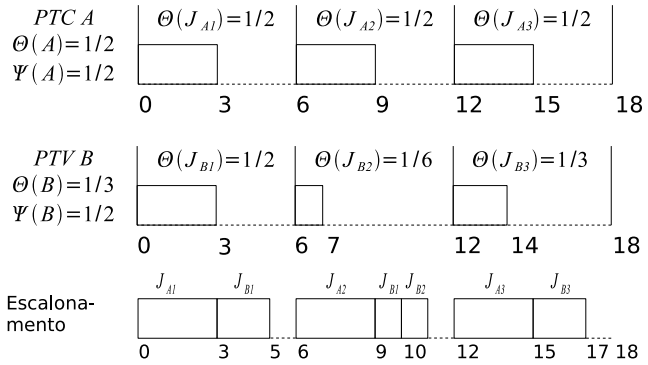


Figura 11: Exemplo de 2 tarefas sendo escalonadas com proteção *overrun*.

2.3.5 Considerações Finais

Eclipse/BSD desempenha o papel de fornecer QoS em um sistema de propósito geral, com o objetivo de minimizar mudanças nas aplicações. Para isso foi criado um sistema de reservas hierárquico ponderado, com controle de admissão, baseado na criação de um sistema de arquivos, a ser utilizado pelas aplicações. Em sistemas embarcados, a centralização do controle e gerenciamento de QoS em um sistema de arquivos pode não ser representativo das necessidades das aplicações, pois isso acarretaria em um *overhead* desnecessário às requisições de QoS.

QoSOS fornece um *framework* genérico para provisão de QoS em sistemas operacionais. Este conjunto de *frameworks* atende as necessidades genéricas de um sistema que requer QoS como, por exemplo, controle de admissão e adaptabilidade. Contudo, por ser um conjunto de modelos que visam atender de maneira completa o problema de provisão de qualidade, sua complexidade se mostra demasiada para sistemas embarcados.

A arquitetura para sistemas MPSoC baseados em redes intra-chip proposta em [14] evidencia as diferenças entre sistemas de propósito geral e sistemas embarcados. Em uma aplicação específica e real foi demonstrada a importância de QoS em sistemas embarcados devido a sua crescente complexidade. Também foi aplicado o conceito de gerenciador de QoS, mas dividido entre gerenciador local e global.

R-EDF fornece uma forma eficaz de garantir reservas de processamento para tarefas de tempo-real, porém não contempla tarefas *hard real-time*. Existe, também, a possibilidade de otimizar determinados casos de processamento. Tais possibilidades serão focadas com maior ênfase no Capítulo 3.

3 Arquitetura do Sistema

Devido o aumento da complexidade em sistemas embarcados, o surgimento de tecnologias que agilizem o desenvolvimento de aplicações, com alta performance, exigem que abstrações sejam usadas. Os sistemas operacionais abstraem a camada de *hardware* das aplicações. Como a provisão de QoS está altamente vinculada aos recursos fornecidos pelo SO, como, por exemplo, controle sobre filas de comunicação, optou-se por acoplar o sistema de provisão de QoS ao sistema operacional, mais especificamente, no escalonador de processos.

Visando atingir o objetivo deste trabalho, foi implementada um arquitetura de *hardware* embarcado, um SoC. Um sistema operacional embarcado foi estudado e adaptado à essa arquitetura. Para prover QoS no escalonador do SO, estudou-se o algoritmo de escalonamento com reservas R-EDF. Propõe-se um novo algoritmo, baseado no R-EDF, que traz modificações ao mesmo para melhorar a performance e adicionar suporte a aplicações *hard real-time*, foco deste trabalho. Esse novo algoritmo é denominado ER-EDF. Assim, obteu-se um sistema operacional embarcado executando em um SoC, com funcionalidades de provimento de QoS no escalonamento de processos. Com isso, espera-se contribuir para o provimento de QoS fim-a-fim, em uma parte importante do processo, que é o escalonamento de aplicações.

A capacidade do *hardware* disponível para o desenvolvimento do trabalho é limitada. Assim, agregando tal limitação à limitação de tempo para desenvolvimento, foi implementado somente um SoC monoprocessado. Entretanto, a arquitetura de software é modelada de forma a permitir uma futura expansão para sistemas multiprocessados fornecendo, assim, a escalabilidade desejada para o sistema.

Para viabilizar a execução de testes, foram criadas duas ferramentas: QoS Tester e QoS Analyser. A primeira é encarregada da comunicação com o sistema operacional executando na arquitetura de *hardware*. A segunda é encarregada de mapear e analisar dados capturados, oferecendo estatísticas sobre as tarefas executadas.

A seguir serão mostradas as arquiteturas de *hardware* e de *software* implementadas. Serão ilustradas as soluções criadas para o provimento de QoS e, também, as ferramentas desenvolvidas para a execução e análise de testes.

3.1 Arquitetura de *Hardware*

3.1.1 Prototipação

Para viabilizar a prototipação do processador, foi utilizada a placa Spartan-3 Starter Board [18]. Essa placa possui disponíveis: uma memória, do tipo SRAM, de 1 MiB[1], 4 *displays* de 7 segmentos, *leds*, porta serial e comunicação JTAG para configuração.

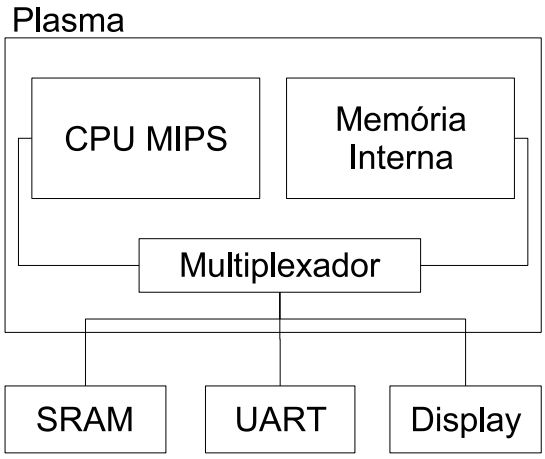


Figura 12: Arquitetura de *Hardware*.

Na figura 12 está ilustrado como são conectados os diferentes periféricos implementados na arquitetura. A CPU Plasma está interconectada através de um multiplexador à memória interna, memória externa, serial e *display* de sete segmentos. A memória interna, implementada em *brams* possui 8 KiB e é inicializada em tempo de compilação através de um processo automatizado. A memória externa, uma SRAM , possui um MiB e é inicializada através da comunicação serial. O *display* é utilizado para exibição de dados ao usuário sem o *overhead* de tempo imposto pela serial, que é o principal meio de comunicação com o mundo externo à arquitetura.

3.1.2 Processador Plasma

A arquitetura de *hardware* implementada contempla o processador *soft-core* Plasma. Plasma implementa o conjunto de instruções MIPS[2], é de código aberto e, portanto, é flexível para as mudanças necessárias para o correto funcionamento do sistema operacional.

O processador Plasma[3] é um projeto simples que implementa somente um subconjunto de instruções da arquitetura MIPS. Por exemplo, não está implementado o coprocessador para processamento de números de ponto flutuante. Assim, durante a codificação, não é permitido

[1]Norma IEEE 1541 (1 MiB = 2^{20} bytes)
[2]http://www.mips.com/
[3]Disponível em http://www.opencores.org/projects.cgi/web/mips/overview

utilizar variáveis do tipo *float* ou *double*, o que limita a precisão de cálculos tanto dentro de aplicações quanto na implementação do SO.

Em adição às limitações mencionadas, no Plasma não são implementadas determinadas instruções de acesso desalinhado à memória. Assim, para utilizá-lo, fez-se necessário gerar um conjunto de ferramentas de desenvolvimento para a plataforma MIPS, que não fizessem uso das instruções de acesso desalinhado: LWL, SWL, LWR e SWR. Utilizou-se compiladores e ferramentas GNU: GCC[4] e Binutils[5]; e uma biblioteca de funções básicas para programação C denominada *Newlib*[6]. Modificações foram feitas nesses programas para que gerassem código-objeto compatível com a arquitetura MIPS, excluindo as instruções não implementadas.

Para o controle de geração de interrupção de *timer*, um registrador de 32 bits é implementado interno à arquitetura do Plasma. Ele é incrementado a cada ciclo de *clock* e pode ser acessado através de uma leitura de memória. É também utilizado para captura de *timestamps* para temporização de seções da execução.

Uma interrupção é gerada quando o bit 14 desse registrador atinge o estado 1. Este registrador funciona, também, como um *timer*. O processador plasma é prototipado para executar a uma freqüência de 25MHz, o que serve de base para o cálculo do período e freqüência da interrupção de *timer*. Quando o bit 14 entra no estado 1, esse estado perdura por 2^{14} ciclos. Então entra no estado 0 que possui a mesma duração. Assim, a soma desses dois estados resulta no número de ciclos de um período: $2^{14} + 2^{14} = 2^{15}$. Para verificar a eficácia desse mapeamento de tempo, uma aplicação simulando um cronômetro foi implementada e seus resultados comparados com um cronômetro comum. Nesta verificação, ambos relógios prosseguiram a contagem de tempo sincronamente. O cálculo em tempo do período em milisegundos é dado a seguir:

$$
\begin{aligned}
periodo &= \frac{ciclos\ do\ periodo}{\frac{CLOCK}{1000}} \\
periodo &= \frac{2^{15}}{25000} \\
periodo &= 1.31072ms
\end{aligned}
$$

Existe a possibilidade de diminuir o tempo do período alterando o bit que é usado para a ativação da interrupção. Por exemplo, se trocarmos o bit de 14 para 13, obtém-se um período de 2^{14} ciclos, i.e., $0.65536ms$. Entretanto, escolheu-se um período maior que $1ms$ para gerar menor *overhead* de SO e, ainda assim, oferecer resolução compatível com sistemas de tempo-real.

O Plasma conta com um emulador de instruções capaz de executar códigos-objeto. Esta ferramenta é extremamente útil para depuração de código pois permite a execução individual de cada instrução. Entretanto, não implementa interrupções, o que impossibilita algumas funci-

[4] http://gcc.gnu.org/
[5] http://www.gnu.org/software/binutils/
[6] http://sources.redhat.com/newlib/

onalidades do SO, e.g., preempção. Assim, parte do desenvolvimento deve ser feito diretamente com o processador instanciado em *hardware*.

3.2 Sistema Operacional

Escolheu-se o sistema operacional EPOS [19]. EPOS é um sistema operacional orientado à aplicação [20], i.e., se adapta automaticamente aos requisitos da aplicação que o usuário elabora. As principais vantagens da utilização desse sistema na implementação da arquitetura são:

- Código fonte disponível livremente.
- Implementado em C++, uma linguagem altamente documentada e eficaz.
- Orientado a objetos: EPOS é implementado utilizando orientação a objetos como paradigma de programação, o que facilita a compreensão do código fonte e suporta o modelo de alta configurabilidade a que ele se propõe.
- Concebido para aplicações dedicadas.
- Portado para diversas arquiteturas (e.g., Leon, x86).

O sistema operacional EPOS foi portado para arquitetura MIPS, em específico para o microprocessador Plasma. EPOS possui três estágios de inicialização: *boot*, *setup* e *system*. A parte de *boot* é responsável pelas configurações iniciais do sistema e carga da imagem do restante do sistema via porta serial. A parte de *setup*, por sua vez, é responsável por inicializar configurações globais. A parte de *system* possui um conjunto de bibliotecas e contém a aplicação do usuário. As partes de *boot* e *setup* são descartadas após o processo de inicialização. Assim, somente a parte *system* existe residente em memória.

Para a execução de aplicações de tempo-real, um sistema de escalonamento de tarefas de tempo-real foi implementado. Como os algoritmos R-EDF e ER-EDF derivam do EDF, primeiramente o SO EPOS foi adaptado para utilizar o EDF como escalonador. Para viabilizar a implementação do EDF no SO EPOS, algumas mudanças estruturais no escalonador existente foram modificadas como, por exemplo, o modo como a troca de contexto é realizada.

A implementação do EDF exigiu um mecanismo de controle de tempo dentro do SO. Para isso foi introduzido o conceito de *ticks*. Cada *tick* corresponde a uma unidade de tempo dentro do SO e é mapeado em uma variável que é incrementada a cada interrupção. Conforme visto anteriormente, o intervalo entre interrupções é de $1,31ms$. Assim, cada unidade de tempo (*tick*) possui $1,31ms$.

Durante a criação de uma tarefa, seu período é convertido em *ticks*. Por exemplo, uma tarefa de $50ms$ é convertida em 38 *ticks* resultando, assim, em um período real de $38 \times 1,31 =$

$49.78ms$. Seu *deadline* também é transformado em *ticks*. Na Figura 13 é ilustrado o exemplo de um *job* de uma tarefa de período $50ms$. Seu período é convertido a 38 *ticks* e sua utilização ($\theta = 50\%$) é convertida em 19 *ticks*. Supondo que esse *job* foi liberado para execução no décimo *tick* (*release=10*), seu *deadline* se torna o *tick* 48.

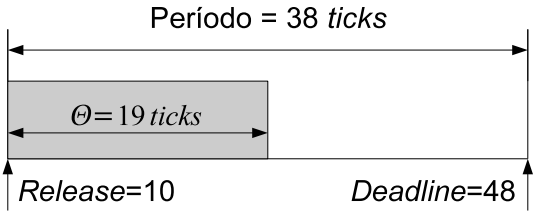


Figura 13: Exemplo de um *job* de período $50ms$ transformado em *ticks*.

3.3 Arquitetura de *Software*

A arquitetura proposta é composta de um gerenciador global de QoS e diversos gerenciadores locais, conforme visto na Figura 14. Caracterizada pelo desacoplamento, a arquitetura cliente-servidor, já utilizada em outros trabalhos [14], mostra-se, a princípio, eficaz para suprir a característica heterogênea dos sistemas MPSoC. Assim, obtém-se, também, a característica de escalabilidade do sistema, bastando adaptar gerenciadores locais compatíveis com o gerenciador global.

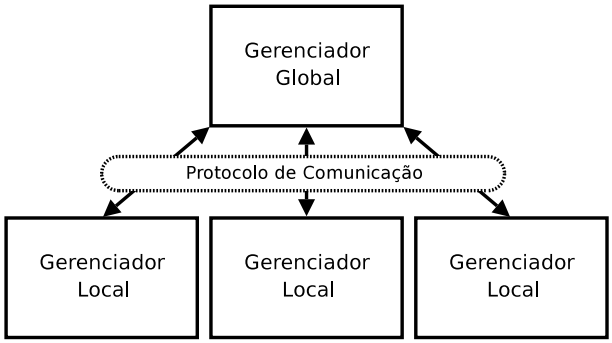


Figura 14: Gerenciadores de QoS do sistema proposto.

Os gerenciadores locais comunicam-se com o gerenciador global que, através dos dados trocados durante a comunicação, atualiza o estado global de alocação de recursos. Como o protótipo implementado é monoprocessado, os gerenciadores global e local são modelados em apenas um gerenciador. Assim, evita-se sobrecarga desnecessária ao sistema.

O gerenciador global é composto por *monitor*, *gerenciador de recursos* e *controle de admissão*, conforme ilustrado na Figura 15. O *monitor* é encarregado de verificar o estado da qualidade de todo o sistema, agregando informações dos *monitores locais*, com o objetivo de garantir os contratos de QoS. O *gerenciador de recursos* gerencia globalmente os recursos disponíveis,

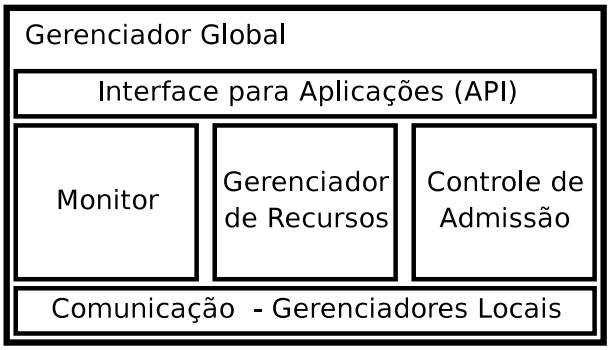


Figura 15: Arquitetura do Gerenciador Global.

interagindo com o *controle de admissão* na verificação da disponibilidade dos mesmos, para garantir a qualidade fim-a-fim. Para comunicação entre gerenciadores globais e locais existe uma camada de comunicação.

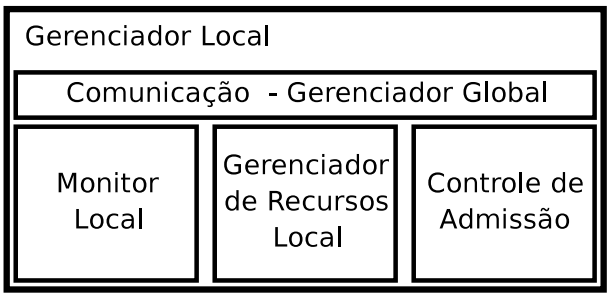


Figura 16: Arquitetura do Gerenciador Local.

Os gerenciadores locais interagem diretamente com os recursos locais de cada elemento de processamento. Ele é também é composto por *monitor*, *gerenciador de recursos* e *controle de admissão*, conforme ilustrado na Figura 16. Diferentemente do gerenciador global, estes atuam sobre as configurações pontuais disponíveis em cada EP. Como exemplo, pode-se citar o controle de largura de banda em um EP que está acoplado a um periférico de acesso a dados. O *monitor* tem a função de monitorar todos os pontos de configuração, atualizando o gerenciador global com as informações capturadas. O *gerenciador de recursos local* tem a função de capturar todos os recursos disponíveis e cadastrar no gerenciador global. O *controle de admissão local* atua sincronamente com o global para o estabelecimento dos contratos de garantias em tempo de execução das aplicações.

3.3.1 Escalonador com Reservas

Para atender as requisições de reservas, o algoritmo R-EDF foi utilizado e modificado para oferecer melhor desempenho em tarefas de tempo-real. Também foi modificado para oferecer suporte a reservas para aplicações *hard real-time*. O processamento é representado como um recurso gerenciável. Assim cabe ao gerenciador local atribuído ao processador gerenciar os processos e se comunicar com o gerenciador global para a manutenção de todo o sistema.

Analisando internamente o algoritmo R-EDF, um problema, ilustrado na Figura 17, foi encontrado. Na figura estão ilustradas duas tarefas PTV. A tarefa A possui reserva $\theta(A) = 1/2$ e a tarefa B possui reserva $\theta(B) = 1/3$. No início da execução, o *job* J_{B1} executa logo após o *job* J_{A1}. Entretanto, o *job* J_{B1} utiliza todo o seu tempo reservado e entra em estado *overrun*. O problema reside nessa reserva restritiva, pois haveria tempo de processamento disponível para o *job* J_{B1} executar nos tempos 3 e 4. Conseqüentemente, ao final dos três primeiros períodos, a tarefa A não consegue terminar, o que poderia ser evitado eliminando a ociosidade encontradas nos tempos 3 e 4.

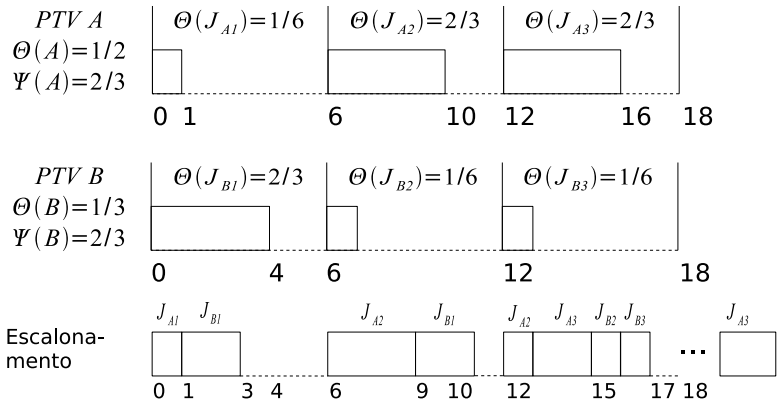


Figura 17: Exemplo de tarefas PTV ilustrando reservas restritivas.

R-EDF não foi concebido para o suporte de tarefas *hard real-time*, pois assume que uma tarefa pode perder seu *deadline*, o que não é desejável. No R-EDF, se uma tarefa entra no estado *overrun*, seu *deadline* não será cumprido. Assim, modificou-se o algoritmo para permitir uma classe de tarefas *hard real-time*.

3.4 Enhanced R-EDF

Enhanced R-EDF (ER-EDF), algoritmo proposto e contribuição deste trabalho, é uma versão aprimorada do algoritmo R-EDF. Como visto anteriormente, R-EDF apresenta restrições no desempenho de escalonamento através do uso de medidas de proteção das tarefas com reservas. Para sanar tais restrições, foram adicionadas modificações que permitem que o tempo ocioso de um período seja utilizado por tarefas do estado *overrun*, i.e., permite-se que uma tarefa com reservas ultrapasse seu tempo reservado.

Também foi adicionado o suporte a reservas para tarefas *hard real-time*. Isto foi alcançado através da reserva pelo pior caso [21], i.e., a reserva é realizada pela utilização pico (ψ). Essa reserva é efetivada somente quando a tarefa é marcada para tal. Permite-se assim o escalonamento de tarefas *hard* e *soft real-time* em um mesmo escalonador.

3.4.1 Controle de Admissão

O controle de admissão sofreu modificações para contemplar tarefas *hard real-time*. Cada *task*, ao ser criada, além de informar valores das utilizações, informa se a tarefa é *hard*. O algoritmo modificado é apresentado a seguir.

Passo 1 Inicialmente a capacidade de tempo-real C_{RTp} e capacidade pico PC_{RTp} de cada processador p são inicializadas com 0 $(1 \leqslant p \leqslant M)$, e a capacidade *time-sharing* C_{TS} é inicializada com M.

Passo 2 Uma tarefa de tempo-real com utilização θ e utilização pico ψ efetua requisição de reserva:

Se a tarefa é *hard* **então**: (reserva pela utilização pico ψ)

Se a capacidade *time-sharing* pode ser reduzida para admitir essa tarefa $C_{TS} - \psi \geqslant \beta$, e um processador p $(1 \leqslant p \leqslant M)$ pode atingir o requerimento $C_{RTp} + \psi \leqslant 1$

Então a tarefa é admitida e atribuída ao processador p, com: $C_{RTp} = C_{RTp} + \psi$; $PC_{RTp} = PC_{RTp} + \psi$; $C_{TS} = C_{TS} - \psi$

Senão a tarefa é rejeitada.

Senão: (reserva pela utilização θ)

Se a capacidade *time-sharing* pode ser reduzida para admitir essa tarefa $C_{TS} - \theta \geqslant \beta$, e um processador p $(1 \leqslant p \leqslant M)$ pode atingir o requerimento $C_{RTp} + \theta \leqslant 1$

Então a tarefa é admitida e atribuída ao processador p, com: $C_{RTp} = C_{RTp} + \theta$; $PC_{RTp} = PC_{RTp} + \psi$; $C_{TS} = C_{TS} - \theta$

Senão a tarefa é rejeitada.

Passo 3 **Se** uma tarefa de tempo-real com utilização θ e utilização pico ψ, atribuída a um processador p, libera sua reserva, **então**:

Se a tarefa é *hard* **então** $C_{RTp} = C_{RTp} - \psi$; $PC_{RTp} = PC_{RTp} - \psi$; $C_{TS} = C_{TS} + \psi$.

Senão $C_{RTp} = C_{RTp} - \theta$; $PC_{RTp} = PC_{RTp} - \psi$; $C_{TS} = C_{TS} + \theta$.

Na Tabela 1 é exemplificado as diferenças entre R-EDF e ER-EDF no momento do controle de admissão. Para tarefas de melhor esforço é determinado uma reserva de 10%, i.e., $\beta = 0.10$. As seguintes tarefas efetuam o requerimento:

1. **Tarefa 1**: $\theta = 0.4$, $\psi = 0.6$, *hard real-time*.
2. **Tarefa 2**: $\theta = 0.3$, $\psi = 0.4$, *soft real-time*.
3. **Tarefa 3**: $\theta = 0.1$, $\psi = 0.1$, *soft real-time*.

Neste exemplo, três tarefas fazem, seqüencialmente, o pedido no controle de admissão. A primeira tarefa requer uma reserva *hard real-time*, suportado apenas pelo ER-EDF, enquanto que as outras duas requerem *soft real-time*. A primeira linha da tabela mostra o estado inicial do controle de admissão. A primeira tarefa, representada na segunda linha, é aceita em ambos algoritmos. Neste momento, ER-EDF efetuou a reserva pela utilização pico ($\psi = 0.6$) enquanto que R-EDF efetuou pela utilização ($\theta = 0.4$).

Tarefa	R-EDF†	ER-EDF*	Autorizado	*Overloaded*
-	$C_{RT} = 0.0$ $PC_{RT} = 0.0$ $C_{TS} = 1.0$	$C_{RT} = 0.0$ $PC_{RT} = 0.0$ $C_{TS} = 1.0$	-	Não
1	$C_{RT} = 0.4$ $PC_{RT} = 0.6$ $C_{TS} = 0.6$	$C_{RT} = 0.6$ $PC_{RT} = 0.6$ $C_{TS} = 0.4$	Sim†/Sim*	Não
1+2	$C_{RT} = 0.7$ $PC_{RT} = 1.0$ $C_{TS} = 0.3$	$C_{RT} = 0.9$ $PC_{RT} = 1.0$ $C_{TS} = 0.1$	Sim†/Sim*	Sim $PC_{RT} > 1 - \beta$ $1.0 > 0.9$
1+2+3	$C_{RT} = 0.8$ $PC_{RT} = 1.1$ $C_{TS} = 0.2$	$C_{RT} = 1.0$ $PC_{RT} = 1.1$ $C_{TS} = 0.0$	Sim†/Não*	Sim $PC_{RT} > 1 - \beta$ $1.1 > 0.9$

Tabela 1: Tabela exemplificando controle de admissão de R-EDF e ER-EDF.

Então, a segunda tarefa faz o requerimento e o sistema entra, para ambos algoritmos, no estado *overloaded*. A última coluna da tabela mostra o cálculo realizado para determinar o estado *overloaded*. Quando a terceira tarefa tenta iniciar sua execução, o controle de admissão do R-EDF permite, enquanto que no ER-EDF a tarefa é rejeitada. O critério que determina a rejeição é mostrado no seguinte cálculo:

$$\begin{aligned} C_{TS} - \theta &\geqslant \beta \\ 0.1 - 0.1 &\geqslant 0.1 \\ 0.0 &\geqslant 0.1 \end{aligned}$$

3.4.2 Escalonamento

No ER-EDF, o escalonamento foi aprimorado para fazer melhor uso do tempo de processamento. Para isso, foram feitas modificações para liberar o tempo ocioso de processamento para processos que estejam em *overrun*. As principais modificações englobam:

1. Evitar que a tarefa entre no estado *overrun* quando não existe outra para tarefa pronta para execução;

2. Ao término de um *job*, retirar tarefa de deadline mais próximo do estado *overrun* se não houver tarefas prontas para executar.

Analogamente ao R-EDF, ER-EDF somente ativa o mecanismo de proteção quando se encontra no estado sobrecarregado. Assim, quando o sistema está sobcarregado, ER-EDF se comporta da mesma forma que o algoritmo EDF original. O Algoritmo ER-EDF é descrito a seguir.

Passo 1 Selecione uma tarefa para execução:

1. **Se** alguma tarefa de tempo-real está pronta, **então** seleciona a tarefa cujo último *job* liberado para a lista LJD tenha o *deadline* mais próximo e execute seus *jobs*.
2. **senão se** há alguma tarefa no estado *overrun*, **então** seleciona a tarefa que está em *overrun* cujo último *job* liberado para a lista LJD tenha o *deadline* mais próximo e execute seus *jobs*.
3. **senão** invoque o escalonador de tarefas de melhor-esforço.

Passo 2 O escalonador espera até a próxima unidade de tempo:

1. **Se** a tarefa que está executando termina a execução de seus *jobs*, i.e., a LJD está vazia, **então** ela entra no estado *esperando*.
2. **Senão se** o sistema está sobrecarregado e ainda há *jobs* na LJD da tarefa corrente e a tarefa utilizou seu tempo reservado, **então**

   Se há tarefas prontas para escalonar, **então** ela entra o estado **overrun**.

   Senão se a utilização executada é maior ou igual a $(1-\beta)$, **então** ela entra o estado **overrun**.

   senão continua a execução.
3. Verificar todas tarefas pelo início de um novo período e colocá-las no estado **pronto**.

Passo 3 Vá para o passo 1.

Na Figura 18 está ilustrado o diagrama de estados representando o novo algoritmo ER-EDF. As principais mudanças se encontram na transição entre o estado *overrun* e o estado *pronto*. Uma tarefa pode permanecer executando no processador mesmo quando ultrapassa o tempo reservado, atrasando, assim, a entrada no estado *overrun*. Também existe a possibilidade de sair do estado *overrun* para executar.

Na Figura 19 é ilustrada a execução de duas tarefas PTV conforme explanado na Subseção 3.3.1. Enquanto que, na execução do algoritmo R-EDF, o *job* J_{B1} é colocado no estado *overrun*, ER-EDF verifica que não há tarefas para executar e permite que o *job* execute nos tempos 3 e 4 mesmo ultrapassando seu tempo reservado. No tempo 15, o *job* J_{A3} entra no estado *overrun* permitindo que o *job* J_{B3} execute. Logo após a execução de J_{B3}, o *job* J_{A3} sai do estado *overrun* e executa no tempo restante.

O Gerenciador Local está intrinsecamente ligado ao algoritmo de escalonamento e seu método de controle de admissão. Assim, mapeando os componentes do Gerenciador Local ao algoritmo de escalonamento, pode-se especificar:

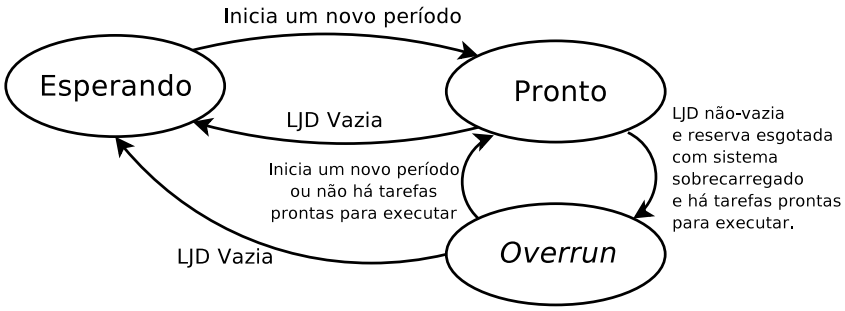


Figura 18: Diagrama de estados do algoritmo ER-EDF.

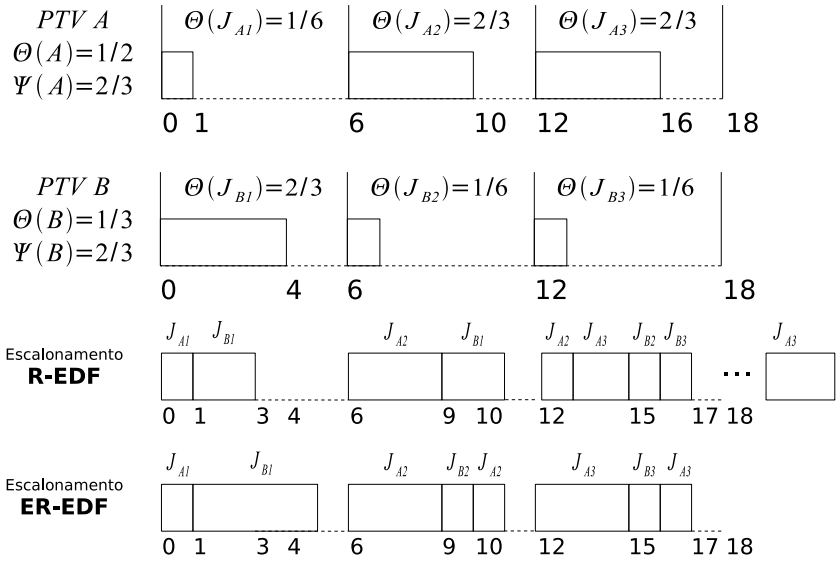


Figura 19: Exemplo de tarefas PTV ilustrando reservas restritivas e solução do algoritmo ER-EDF.

- **Controle de Admissão**: é o próprio controle de admissão do ER-EDF.
- **Gerenciador de Recursos**: neste, mapeia-se o processador como sendo um recurso gerenciável e, assim, pode ser reservado e oferecer garantias através de método especificado pelo algoritmo de escalonamento que, neste caso, é o ER-EDF.
- **Monitor**: mapeia-se para um medidor de tempo ocioso de CPU. Este incrementa um contador de tempo.

3.5 Fluxo de Geração, Execução e Análise de Testes

Para a validação dos algoritmos de escalonamento propostos, implementou-se um fluxo de geração, execução e análise de testes. QoS Tester e QoS Analyser são duas ferramentas criadas para automatizar esse fluxo. A geração e execução de testes é de responsabilidade do QoS Tester. Já a parte de análise é realizada pela ferramenta QoS Analyser.

Na Figura 20 está ilustrado o fluxo de execução de testes. Primeiramente o usuário informa

os dados para geração de testes, i.e., descrição de tarefas e valores das distribuições (1). A partir desses dados, testes são gerados utilizando distribuições matemáticas (2). Então o executor de testes, comunicando com a plataforma de *hardware* dá seqüência à execução dos testes (3). Para cada imagem de EPOS (4) uma bateria de testes é executada. Então os dados gerados por esses testes são capturados e armazenados (5). Utilizando os dados, QoS Analyser mapeia para estruturas de dados (6) que depois serão consultadas para a geração de estatísticas (7). A seguir serão aprofundados os principais conceitos e funcionamento destas duas ferramentas.

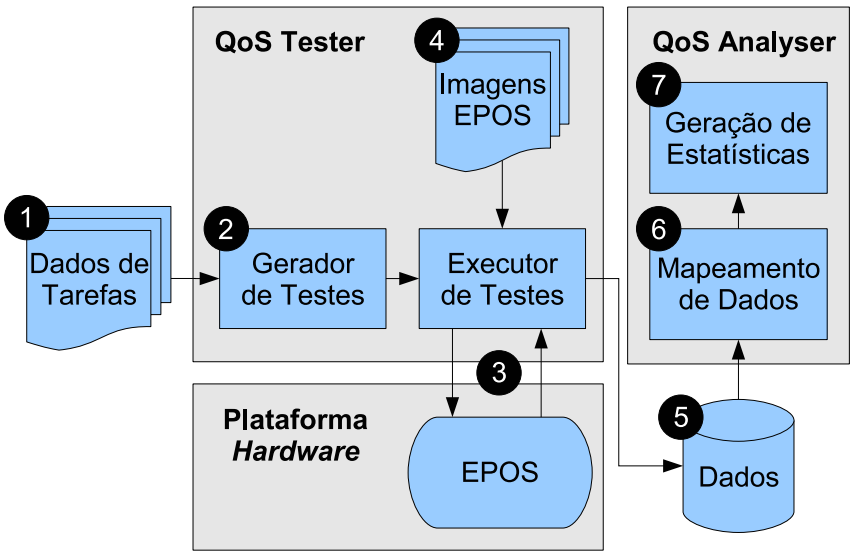


Figura 20: Fluxo de execução e geração de dados para análise de testes.

3.5.1 QoS Tester

QoS Tester é a ferramenta responsável por capturar dados gerados pela execução de tarefas na arquitetura de *hardware* desenvolvida. Esta ferramenta foi concebida devido a limitação de memória da arquitetura de *hardware* e, também, para automatização de testes. Como a memória é limitada para a captura de dados de execução, um método automatizado para capturar os dados foi criado. Parte da implementação encontra-se dentro do sistema operacional que executa no sistema embarcado, denominado *client*, e parte na máquina servidora, denominado *host*. *Client* executa as tarefas de tempo real e envia os dados gerados através da porta serial para a máquina *host*. A lista de pré-requisitos para o correto funcionamento desta aplicação é exibido na Tabela 2.

Dentro do sistema operacional EPOS, implementou-se uma classe que armazena eventos gerados durante a execução. Cada evento possui os seguintes campos:

- **Tipo**: tipo do evento. Cada tipo de evento possui um identificador, um código e sua função, descritos a seguir:
  - INT_BEGIN (0): início de uma interrupção;

Pré-requisito	Função
Sistema Operacional	A aplicação foi implementada no sistema operacional Linux. Entretanto, o esforço para portar para outro SO depende somente do acesso aos demais pré-requisitos como, por exemplo, acesso a porta serial.
Serial	Uma biblioteca foi utilizada para o acesso a serial. Denominada Commoncpp, disponível em http://gnu.org/software/commoncpp. Esta biblioteca implementa acesso simplificado a serial mapeando-a em *streams* da linguagem de programação C++.
Gerador de Números Randômicos	Para a geração de números randômicos foi utilizada a biblioteca Boost, disponível em http://www.boost.org/. Esta biblioteca implementa a geração de números através de parâmetros especificados pelo programador. As distribuições utilizadas foram a constante e linear.
Impact	Para carregar o *bistream* contendo o processador Plasma para a placa utilizada, faz-se necessária uma ferramenta específica denominada Impact. Esta ferramenta é parte integrante do pacote de *software* para desenvolvimento de *hardware* da empresa Xilinx Inc.
EPOS	Três imagens com o SO EPOS devem ser geradas para serem enviadas para serem executados no processador. Cada imagem deve conter uma implementação dos algoritmos testados: EDF, R-EDF e ER-EDF.

Tabela 2: Pré-requisitos da aplicação QoS Tester.

  - INT_END (1): fim de uma interrupção;
  - EXEC_BEGIN (2): início da execução de uma tarefa;
  - EXEC_END (3): fim da execução de uma tarefa;
  - JOB_RELEASE (4): momento em que um *job* é liberado para execução;
  - JOB_END (5): momento que que um *job* termina;
  - JOB_DEADLINE (6): informa o *deadline* do *job*;

- **Time**: é um *timestamp* capturado do registrador de *timer* implementado em hardware. Também pode armazenar o *deadline* quando Tipo=JOB_DEADLINE.

- **Id**: identificador da *thread* que está executando.

- **Tick**: *tick* em que ocorreu o evento.

Diversas chamadas para a função que armazena eventos foram colocadas em pontos de início e fim de medição. Os pontos de chamada das funções são:

- **Interrupção**: um evento no início (INT_BEGIN) do tratamento e um no fim (INT_END);

- **Liberação de um *Job***: No momento da liberação de um *job* em uma tarefa, são gerados dois eventos: JOB_RELEASE e JOB_DEADLINE. O primeiro indica o momento em que a aplicação liberou o *job* armazenando o tempo corrente. O segundo armazena no campo *time* o *deadline* do *job* liberado.

- **Execuções**: ao fim de uma interrupção, inicia-se uma nova execução, marcada pelo parâmetro EXEC_BEGIN. Uma execução pode terminar com o início de uma interrupção (EXEC_END) ou com o fim de um *job*. EXEC_END é sempre seguido de um INT_BEGIN e um INT_END é sempre seguido de um EXEC_BEGIN.

- **Fim de um *Job***: um evento é criado ao fim de um job.

Os dados gerados durante a execução de testes são armazenados em um *buffer* na memória interna do *client*. Quando o *buffer* alcança seu limite máximo de armazenamento, os dados nele contidos são transmitidos via porta serial e armazenados em arquivo pela aplicação na máquina *host*. Um protocolo simplificado, baseado em caracteres de controle, foi implementado para a comunicação via serial. Enquanto um caractere de controle não é transmitido, os caracteres recebidos são impressos na tela do *host*. Esses caracteres impressos na tela são os gerados pela aplicação dentro do sistema operacional e, também, pelo próprio SO. Durante uma execução de testes várias transmissões podem ocorrer. Estas são automaticamente tratadas através do protocolo.

Em cada teste realizado, dados são gerados para cada tarefa com base em parâmetros fornecidos pelo usuário. Para cada tarefa devem ser fornecidos 5 parâmetros: *jobs*, período, *hard/soft* e dois parâmetros para geração de dados. *Jobs* indica o número de *jobs* a serem executados e período é o valor do período em milisegundos. A geração de um vetor de utilizações dá-se por duas distribuições: linear e constante. Para gerar números através da distribuição linear, os últimos dois parâmetros da tarefa devem informar os números mínimo e máximo que serão gerados. Já a geração da distribuição requer apenas o valor da constante desejada. Para cada tarefa é gerado um vetor contendo a utilização de cada *job* a ser executado.

Para automatizar o processo de testes, foi implementado um processo de execução de testes. Os passos desse processo são:

1. *Host* gera os dados (vetor de utilizações) de execução baseado nas entradas do usuário;

2. *Host* carrega o *bitstream* contendo o processador Plasma e o processo de *boot*;

3. *Client* aguarda envio da imagem a ser executada;

4. *Host* envia imagem com o sistema operacional EPOS;

5. *Client* aguarda o envio do número de tarefas a serem criadas;

6. Para cada tarefa, receber *período, utilização, utilização pico* e vetor de utilizações para *jobs.*

7. *Client* cria as tarefas e inicia a execução;

8. Enquanto houver dados para receber, *host* recebe e armazena os dados.

O processo de transferência de dados via porta serial é realizado no item 8 do processo supracitado. Este é um processo custoso em tempo pois é limitado pela velocidade da porta serial. Para que o tempo utilizado não fosse erroneamente computado para o *job* de alguma tarefa, criou-se um mecanismo que somente permite que a transferência seja realizada durante o tratamento de interrupção, i.e., entre INT_BEGIN e INT_END. Assim, garante-se que o tempo gasto na transferência será adicionado somente ao tempo de interrupção.

Durante transferência de dados, as interrupções são desligadas e, assim, *ticks* não são incrementados. Isso resulta em uma parada de tempo virtual, pois, internamente ao SO, o tempo é contado em *ticks*, que são incrementados a cada interrupção.

Automatizando ainda mais o método de teste, uma imagem contendo o SO EPOS foi gerada para cada algoritmo de escalonamento testado. Assim, o processo de teste explanado acima é executado uma vez para cada algoritmo. Com essa automatização, todos algoritmos são testados em apenas uma execução.

Para cada teste de algoritmo, os dados capturados são armazenados em um arquivo. Abaixo, segue um exemplo da saída gerada pelo *client*:

```
=275431905=5=3=0
```

Contendo campos separados pelo símbolo ”=”, cada evento possui quatro informações, respectivamente, *time*, *tick*, *id* e *tipo*. Neste exemplo, a tarefa de identificador 3 recebeu uma interrupção (*tipo* = 0 = *INT_BEGIN*), no *tick* 5. O *timer*, contador de 32 bits em *hardware*, possuia o número 275431905 no momento da medição. Essa medição de *timer* permite cálculos de grande precisão de tempo. Diminuindo-se uma medição atual com uma anterior, obtém-se o número de instruções executadas entre as medições, pois o *timer* é incrementado a cada ciclo de *clock* e o processador Plasma executa uma instrução por ciclo de *clock*.

Na Tabela 3 estão demonstrados dados de saída de uma execução exemplo. A primeira coluna demonstra o agrupamento dos eventos e está diretamente relacionada à Figura 21. A segunda coluna numera as linhas. A terceira apresenta os dados capturados pela ferramenta de teste. A quarta coluna traduz o que cada evento significa. A última coluna agrupa os diferentes eventos em blocos fornecendo o tempo, calculado em microsegundos, que cada bloco executou.

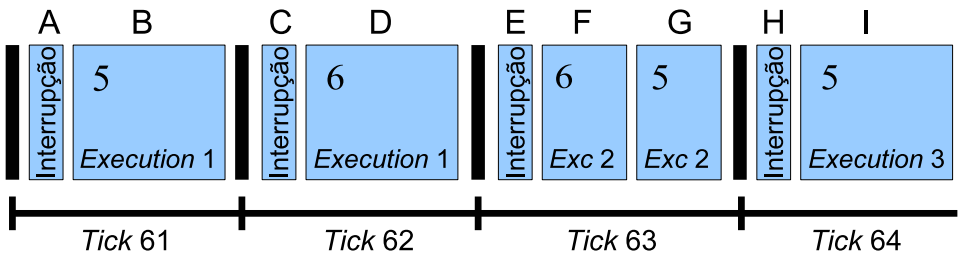


Figura 21: Representação gráfica do exemplo apresentado na Tabela 3.

Na Figura 21 é exemplificado graficamente a mesma execução. No primeiro *tick* – *tick 61* – é realizado o *release* da tarefa de identificador 5 (linha 2 da Tabela 3). Esta tarefa passa, então,

a executar. Na interrupção seguinte, o *tick* é incrementado e a tarefa 6 sofre um *release* (linha 8 da Tabela 3). Como essa tarefa tem *deadline* menor, (*deadline* = 72, linha 9 da Tabela 3) ela passa, então, a executar. Na metade do *tick* seguinte, o *job* da tarefa 6 termina, cumprindo seu *deadline* pois $63 < 72$.

Findando o *job* da tarefa 6, retoma-se a execução da tarefa 5. Esta executa até sofrer uma interrupção que incrementa o *tick* e a mantém executando até seu fim. A tarefa 5 possui 3 execuções que formam um *job*. A tarefa 6, por sua vez, possui duas execuções.

Tabela 3: Exemplo de saída de dados geradas pela execução de tarefas.

Grupo	Linha	Evento	Significado	Duração
A	1	=277233842=60=2=0	Início de interrupção, *tick* = 60.	$30.76\mu s$
	2	=277234293=61=5=4	*Release* da tarefa 5, *tick* = 61.	
	3	=81=61=5=6	*Deadline* da tarefa 5 é o *tick 81*.	
	4	=277234611=61=5=1	Fim da interrupção.	
B	5	=277234638=61=5=2	Início de execução da tarefa 5, *tick* = 61	$1277.78\mu s$
	6	=277266580=61=5=3	Fim de execução da tarefa 5.	
C	7	=277266611=61=5=0	Início de interrupção, *tick* = 61.	$30.76\mu s$
	8	=277267062=62=6=4	*Release* da tarefa 6, *tick* = 62.	
	9	=72=62=6=6	*Deadline* da tarefa 5 é o *tick 72*.	
	10	=277267380=62=6=1	Fim da interrupção.	
D	11	=277267407=62=6=2	Início de execução da tarefa 6, *tick* = 62.	$1277.64\mu s$
	12	=277299348=62=6=3	Fim de execução da tarefa 6.	
E	13	=277299379=62=6=0	Interrupção.	$21.48\mu s$
	14	=277299916=63=6=1		
F	15	=277299943=63=6=2	Início de execução da tarefa 6, *tick* = 63.	$647.44\mu s$
	16	=277316129=63=6=5	Fim de execução e fim do *Job* da tarefa 6.	
G	17	=277316278=63=5=2	Início de execução da tarefa 5, *tick* = 63.	$633.52\mu s$
	18	=277332116=63=5=3	Fim de execução da tarefa 5.	
H	19	=277332147=63=5=0	Interrupção.	$21.48\mu s$
	20	=277332684=64=5=1		
I	21	=277332711=64=5=2	Início de execução da tarefa 5, *tick* = 64.	$1286.92\mu s$
	22	=277364884=64=5=5	Fim de execução e fim do *Job* da tarefa 5.	

3.5.2 QoS Analyser

QoS Analyser é a ferramenta que interpreta os dados capturados pelo QoS Tester. Os dados são mapeados em objetos que, uma vez criados, podem ser consultados para a obtenção de informações como, por exemplo, taxa de perda de *deadlines*.

Na Figura 22 está ilustrado o diagrama de classes da ferramenta. *Processor* é a classe principal que controla todas outras. *Processor* pode possuir uma ou mais *Tasks*. A classe *Task* representa uma tarefa executada no sistema *client*. *Task* possui um ou mais *Jobs* que, por sua vez, possuem uma ou mais *Executions*.

A classe *Processor* é responsável por ler o arquivo de dados, criar as *tasks* e enviar para cada uma seus dados pertinentes. Cada *task* somente recebe dados cujo campo *id* é igual ao seu identificador.

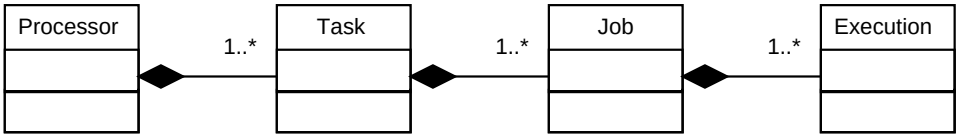


Figura 22: Diagrama de classes da ferramenta QoS Analyser.

Task representa uma tarefa. Contém funções para consultar informações sobre seus *jobs*. A classe *Job* armazena os dados de cada *job*, possuindo funções para consultas sobre o job. Por exemplo, a função `bool met_deadline()` retorna se o *job* teve seu *deadline* cumprido. Dentro da classe *Task*, cada *job* é consultado, formando, assim, a taxa de perda de *deadlines*.

Dentro da classe *Job* têm-se diversas execuções, mapeadas pela classe *Execution*. Uma execução representa uma unidade de tempo executada pelo *job*. Assim, se um *job* possuir 3 execuções, terá executado por, no máximo, 3 unidades de tempo. Cada *Execution* possui variáveis que determinam o início e o fim da execução. Os dados de início e fim são capturados do *timer* em *hardware* dentro do *client*. Como o *timer* incrementa a cada ciclo de *clock*, é possível dizer com precisão quantas instruções cada *job* necessitou para executar. Para obter essa informação em tempo (milisegundos), basta convertê-la.

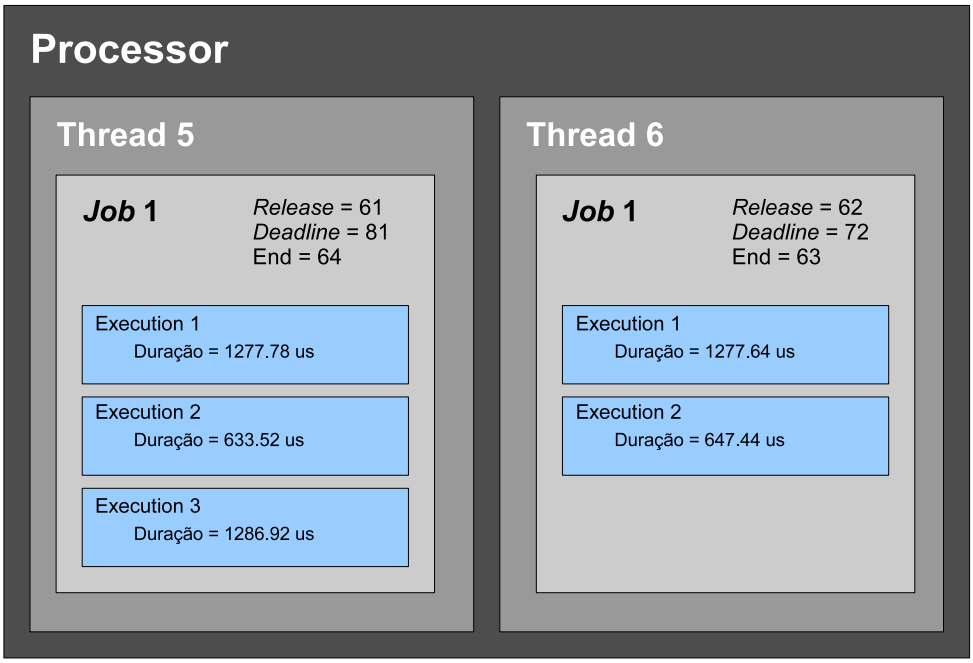


Figura 23: Mapeamento de dados do exemplo apresentado na Tabela 3.

Dentro do SO EPOS, foi implementado um sistema de identificação de tarefas. Cada tarefa recebe um número incremental iniciando em 1. No QoS Analyser, a tarefa 0 foi mapeada com as execuções de interrupção. Assim, pode-se obter dados estatísticos sobre o tratamento de interrupção, e.g., média. A tarefa 0 possui apenas um *job*, onde é instanciada cada execução de tratamento de interrupção. As tarefas 1 e 2 são mapeadas, respectivamente, a aplicação principal e a tarefa *idle*. A aplicação principal é responsável por inicializar as demais tarefas. Assim, as

tarefas são instanciadas dentro da aplicação principal. A tarefa *idle* executa quando não há mais tarefas a serem executadas.

A Figura 23 apresenta o mapeamento do exemplo apresentado anteriormente na Tabela 3. Neste exemplo são criadas duas *Threads* possuindo, cada uma, um *job*. O *job* da *Thread* 5 possui três *Executions*. O *job* da *Thread* 6, por sua vez, possui duas *Executions*. A partir desse mapeamento, é possível calcular algumas informações como, por exemplo, taxa de perda de *deadlines* e período. A taxa de perda de *deadlines* para ambos *jobs* é de 0%, pois, em ambos casos, $End < Deadline$. O período pode ser calculado diminuindo-se *release* de *deadline* ($deadline - release$). Por exemplo, para a *Thread* 5, o período é de 20 *ticks*.

4 Estudo de Caso

Neste capítulo serão apresentados os experimentos realizados para a validação do algoritmo proposto. Os testes foram realizados através da ferramenta QoS Tester, que executa, automaticamente, tarefas previamente estabelecidas. Então a ferramenta QoS Analyser é utilizada para gerar os dados estatísticos aqui apresentados. Serão apresentados 5 testes que demonstram as principais diferenças entre EDF, R-EDF e ER-EDF.

Na Tabela 4 estão explanados os dados que serão apresentados em tabelas para cada teste. Esses dados correspondem às entradas informados pelo usuário da ferramenta QoS Tester. Então, baseados nos vetores de dados gerados, histogramas são criados para visualização do comportamento de cada tarefa.

Tabela 4: Tabela de descrição dos campos para descrição de tarefas.

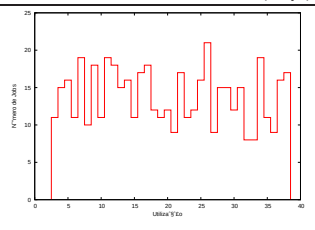
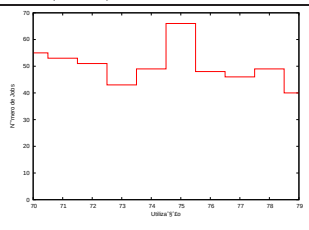
Campo	Função
Jobs	Número de *jobs* executados para a tarefa.
Período	Período em milisegundos e *ticks*.
Distribuição	Tipo da distribuição: linear ou constante.
Valores Distribuição	Valores fornecidos para a geração de testes. Para a distribuição linear os valores representam o mínimo e máximo. Para a constante, o valor indicado é o da constante.
Utilização Média (θ)	Taxa de utilização média de todos os *jobs* da tarefa. Usada para determinar a reserva de tarefas *soft real-time*.
Utilização Pico (ψ)	Taxa de utilização pico de todos os *jobs* da tarefa. Usada para determinar a reserva de tarefas *hard real-time*.
Utilização Média (θ) Total	Soma das utilizações médias. Mede carga total do processador.
Utilização Pico (ψ) Total	Soma da utilização pico de todas as tarefas. Usada para determinar se o sistema se encontra sobrecarregado. Se este campo for maior que 100%, então o sistema está sobrecarregado.
Reserva Total	Reserva total do sistema. Foram realizados teste com e sem reserva *hard real-time*. Assim esse campo exibe a reserva em ambos os casos.
Distribuições	Histograma que ilustra a distribuição gerada. O eixo das abscissas representa a distribuição da utilização entre os *jobs* de uma tarefa. O eixo das ordenadas representa a quantidade de *jobs* com a utilização associada. O gráfico para distribuição constante não é apresentado.

4.1 Teste 1

O primeiro teste visa mostrar diferenças entre o comportamento do escalonamento dos diferentes algoritmos. Neste teste, duas tarefas de período $50ms$ são executadas durante o mesmo

período de tempo. A primeira possui uma distribuição linear onde o mínimo é 3 e o máximo 39, fornecendo uma variação de $\Delta = 36\%$. A segunda, também linear, possui menor variação $\Delta = 9\%$. Para os testes com suporte a tarefas do tipo *hard*, a segunda tarefa foi marcada como sendo *hard real-time*. Na Tabela 5 estão exemplificados as tarefas deste teste.

Tabela 5: Descrição de tarefas - Teste 1.

	Tarefa 1	**Tarefa 2**
Jobs	500	
Período	50ms / 38ticks	
Distribuição	Linear	
Valores Distribuição	3-38%	70-79%
Utilização Média (θ)	21%	75%
Utilização Pico (ψ)	38%	79%
Utilização Média (θ) Total	96%	
Utilização Pico (ψ) Total	117%	
Reserva Total	96% (*soft*), 99% (*hard*)	
Distribuições		

Na Figura 24 está ilustrado o comparativo para perda de *deadlines* em um ambiente sobrecarregado. Cada algoritmo executa o mesmo conjunto de *jobs*. As colunas identificadas com *hard* identificam execuções cuja a reserva *hard real-time* foi ativada. Sem essa identificação, todas tarefas são executadas no modo *soft real-time*.

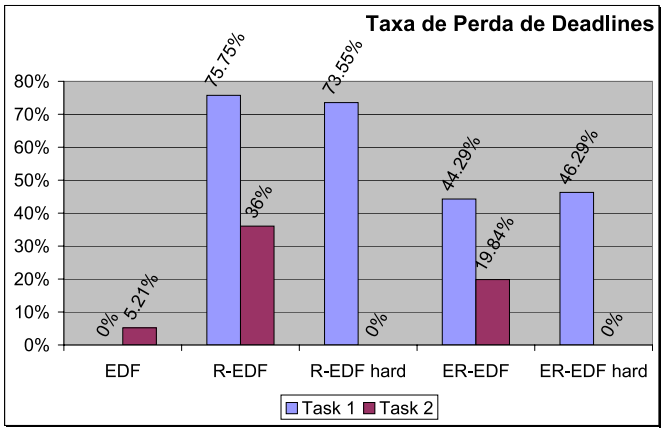


Figura 24: Comparativo para duas tarefas executando com diferentes algoritmos - Teste 1.

EDF apresenta a menor taxa de perda de *deadlines*, considerando tarefas *soft real-time*. Entretanto, EDF não apresenta nenhuma forma de controle sobre qual tarefa irá sofrer perdas. EDF não possui nenhum controle de admissão e, assim, o cenário se tornaria ainda mais imprevisível pois uma tarefa que ultrapassa os limites de capacidade do processador poderia entrar

em execução. Marcando a segunda tarefa com sendo tarefa do tipo *hard real-time*, no R-EDF e ER-EDF, obteve-se 0% de perda de *deadlines*, cumprindo o pré-requisito fundamental para tarefas do tipo *hard*. Nos resultados, observa-se também considerável melhoria de performance na execução do ER-EDF em relação ao R-EDF, tanto para execuções homogêneas *soft real-time* quanto heterogêneas *hard* e *soft real-time*.

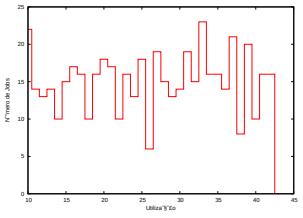
Nas tarefas de reserva *soft real-time* dos algoritmos R-EDF e ER-EDF, observa-se que existe grande perda de *deadlines* quando comparado com o EDF. Isso ocorre pois ambos algoritmos restringem a execução de tarefas pela reserva efetuada por cada um. O controle por reservas resulta em uma inversão de prioridade no momento em que a tarefa entra no estado *overrun*. Assim, um maior número de *deadlines* é perdido. Entretanto, R-EDF e ER-EDF provêem mecanismos para que o desenvolvedor de aplicações determine de que maneira as tarefas serão executadas.

4.2 Teste 2

Neste segundo teste, é apresentada uma situação onde 4 tarefas de tempo-real são executadas. As primeiras 3 possuem utilização constante. Tarefas de utilização constante possuem reserva pico igual a média ($\theta = \psi$). Assim, essa tarefa possui reserva equivalente a tarefas *hard real-time*.

A Tabela 6 apresenta os dados utilizados na geração de testes para as quatro tarefas. A segunda tarefa foi marcada como sendo *hard real-time*. Entretanto sua reserva é realizada de semelhante modo que as tarefas 1 e 3, que também são de distribuição constante. O sistema se encontra sobrecarregado com soma das utilizações pico igual a 115%. A reserva para todas as tarefas é de 100%.

Tabela 6: Descrição de tarefas - Teste 2.

	Tarefa 1	**Tarefa 2**	**Tarefa 3**	**Tarefa 4**
Jobs	500			
Período	50ms / 38ticks			
Distribuição	Constante	Constante	Constante	Linear
Valores Distribuição	26%	21%	26%	10-42%
Utilização Média (θ)	26%	21%	26%	27%
Utilização Pico (ψ)	26%	21%	26%	42%
Utilização Média (θ) Total	100%			
Utilização Pico (ψ) Total	115%			
Reserva Total	100% (*hard* e *soft*)			
Distribuições	Constante	Constante	Constante	

A Figura 25 apresenta os resultados para perda de *deadlines* neste segundo teste. Verifica-se que as tarefas de distribuição constante comportam-se como tarefas *hard real-time* e possuem taxa de perda de *deadlines* igual a 0% nos algoritmos R-EDF e ER-EDF. Observa-se que o uso do algoritmo EDF implica na perda de *deadlines* para todas as tarefas. Enquanto que, para os outros algoritmos, somente tarefas *soft real-time* perdem seus *deadlines*. Nota-se, também, uma ligeira diminuição da perda de *deadlines* na execução do algoritmo ER-EDF quando comparado com R-EDF.

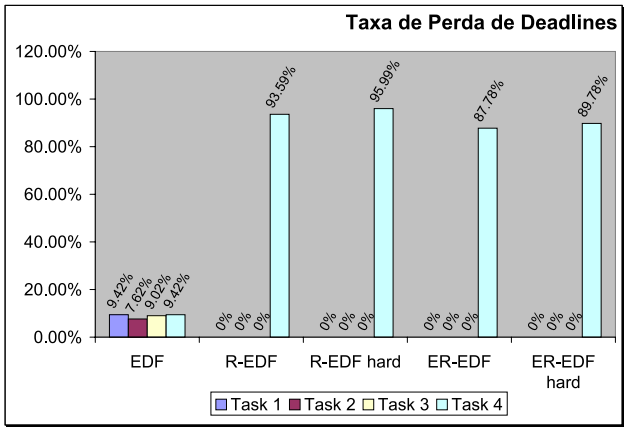


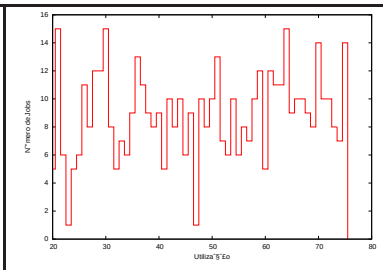
Figura 25: Comparativo para 4 tarefas executando com diferentes algoritmos - Teste 2.

4.3 Teste 3

Neste terceiro teste, uma situação onde o algoritmo EDF se comporta de maneira inesperada é apresentada. A Tabela 7 apresenta a descrição das tarefas. Neste teste, a primeira tarefa é de distribuição constante, a 50%. A segunda tarefa é uma tarefa *soft real-time* de distribuição linear. Entretanto essa segunda tarefa possui grande variação, com $\Delta = 55\%$.

Na Figura 26 são apresentados os resultados de execução para este teste. Nota-se uma perda considerável para as tarefas executadas pelo EDF. A soma das perdas de *deadlines* pelo EDF é de 106.60%, enquanto que para ER-EDF é de 85.37%. Neste caso, o uso do algoritmo ER-EDF mostra-se uma vantagem se a taxa de perda de *deadlines* é um requisito importante. Igualmente a testes anteriores, o ER-EDF apresenta ligeira melhora na perda de *deadlines* da segunda tarefa.

Tabela 7: Descrição de tarefas - Teste 3.

	Tarefa 1	**Tarefa 2**
Jobs	500	
Período	50ms / 38ticks	
Distribuição	Constante	Linear
Valores Distribuição	50%	20-75%
Utilização Média (θ)	50%	49%
Utilização Pico (ψ)	50%	75%
Utilização Média (θ) Total	99%	
Utilização Pico (ψ) Total	125%	
Reserva Total	99%	
Distribuições	Constante	

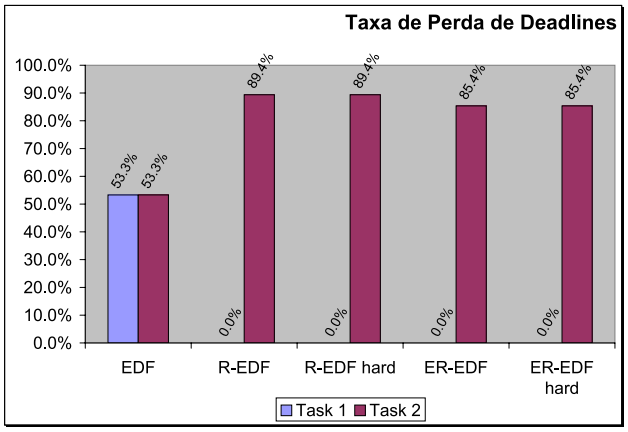
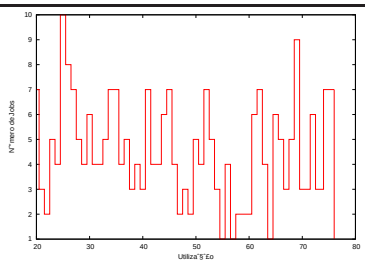


Figura 26: Comparativo para duas tarefas executando com diferentes algoritmos - Teste 3.

4.4 Teste 4

O quarto teste foi elaborado para mostrar que os algoritmos implementados são capazes de gerenciar tarefas de diferentes períodos. Neste teste, duas tarefas de períodos diferentes – $50ms$ e $100ms$ – são executadas. As tarefas possuem distribuições semelhantes ao teste 3, que estão descritas na Tabela 8. Observa-se novamente a atuação da reserva para a primeira tarefa. Também pode-se observar a significativa melhora de 30% na perda de *deadlines* entre os algoritmos R-EDF e ER-EDF.

Tabela 8: Descrição de tarefas - Teste 4.

	Tarefa 1	Tarefa 2
Jobs	500	250
Período	50ms / 38ticks	100ms / 76ticks
Distribuição	Constante	Linear
Valores Distribuição	50%	20-75%
Utilização Média (θ)	50%	49%
Utilização Pico (ψ)	50%	75%
Utilização Média (θ) Total	99%	
Utilização Pico (ψ) Total	125%	
Reserva Total	99%	
Distribuições	Constante	

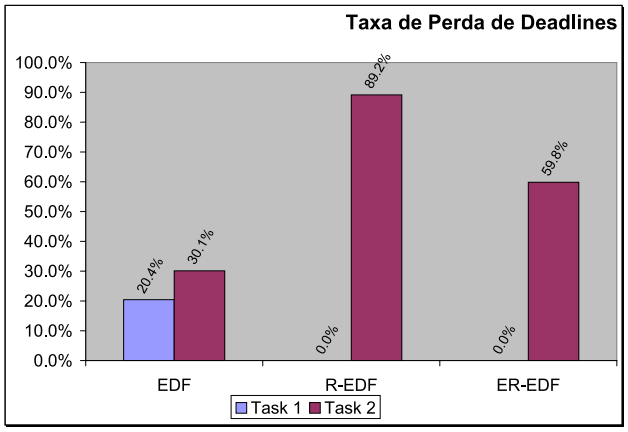


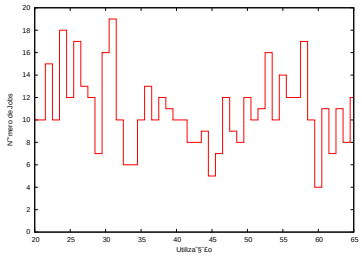
Figura 27: Comparativo para duas tarefas executando com diferentes algoritmos - Teste 4.

4.5 Teste 5

O quinto teste foi elaborado para estudar o comportamento de reserva para tarefas de melhor esforço. Testes anteriores desconsideravam a reserva para essa classe de tarefas considerando sua reserva igual a zero, i.e., $\beta = 0$. Neste teste esta reserva é determinada em $\beta = 5\%$. A Tabela 9 exibe os dados das tarefas executadas durante o teste. Diferentemente de outros testes, a carga total do processador – representada na tabela pela reserva total – é composta da soma de utilização (θ) total e da reserva para tarefas de melhor esforço (β).

Na Figura 28 é apresentado um comparativo entre os algoritmos EDF, R-EDF e ER-EDF representando o tempo máximo de *starvation*, i.e., representa o tempo máximo que as tarefas de melhor-esforço permanecem sem serem executadas. EDF não oferece nenhum tipo de garantias para essa classe de tarefas e obtém o pior resultado do grupo com $874\ ticks$ de tempo

Tabela 9: Descrição de tarefas - Teste 5.

	Tarefa 1	**Tarefa 2**
Jobs	500	
Período	100ms / 76ticks	
Distribuição	Constante	Linear
Valores Distribuição	50%	20-65%
Utilização Média (θ)	50%	42%
Utilização Pico (ψ)	50%	65%
Utilização Média (θ) Total	92%	
Utilização Pico (ψ) Total	115%	
Utilização Melhor-Esforço (β)	5%	
Reserva Total	97%	
Distribuições	Constante	

de *starvation*. R-EDF apresenta o melhor resultado com apenas $38\ ticks$. ER-EDF apresenta o resultado de $286\ ticks$. A Figura 29 exibe um gráfico com a relação entre tempo e número de *ticks* executados pelas tarefas de melhor-esforço. Observa-se que R-EDF mantém uma execução constante durante todo o período de execução. EDF e ER-EDF se aproximam, porém, ER-EDF está freqüentemente acima de EDF, conforme visualizado a partir do $tick\ 20000$.

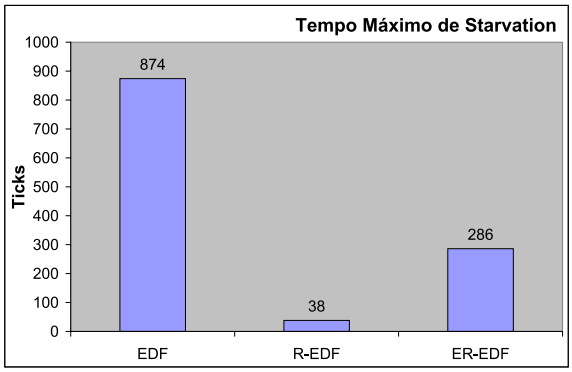


Figura 28: Comparativo para tempo máximo de *starvation* - Teste 5.

A Figura 30 é uma divisão da Figura 29 que demonstra os falhas de atendimento dos requisitos das tarefas de melhor-esforço. Neste gráfico, *starvation* pode ser identificado através das falhas – partes em branco – do gráfico. Observa-se que EDF apresenta visíveis falhas durante a execução. R-EDF, por sua vez, apresenta um comportamento semelhante a um crescimento linear, sem falha visíveis. Já o algoritmo proposto ER-EDF apresenta menos falhas que EDF e mais que R-EDF. Observa-se então que o ganho que ER-EDF obteve sobre o R-EDF em outros testes se dá em detrimento das tarefas de melhor-esforço.

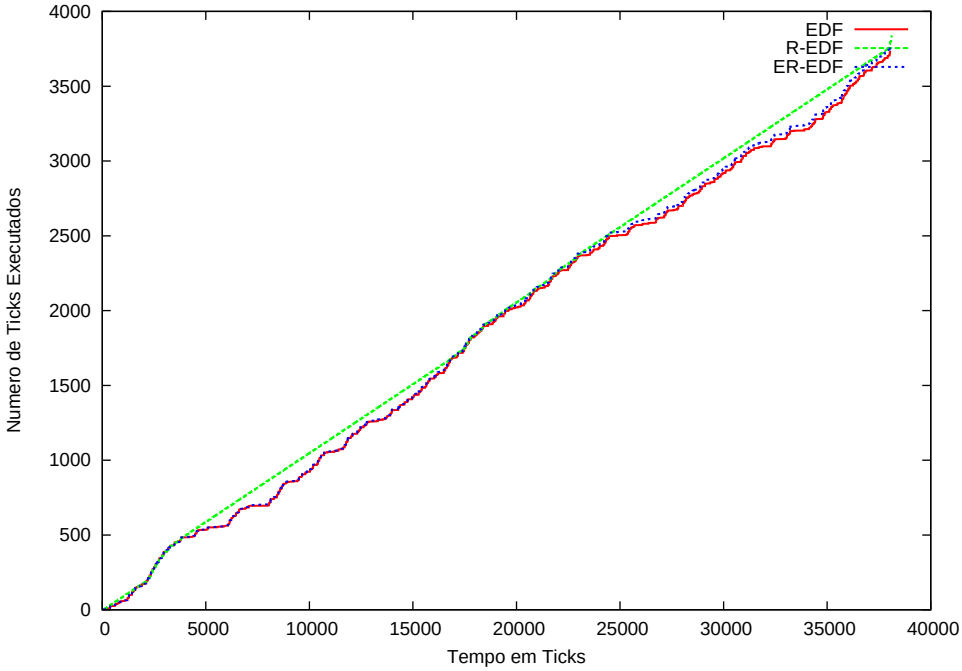
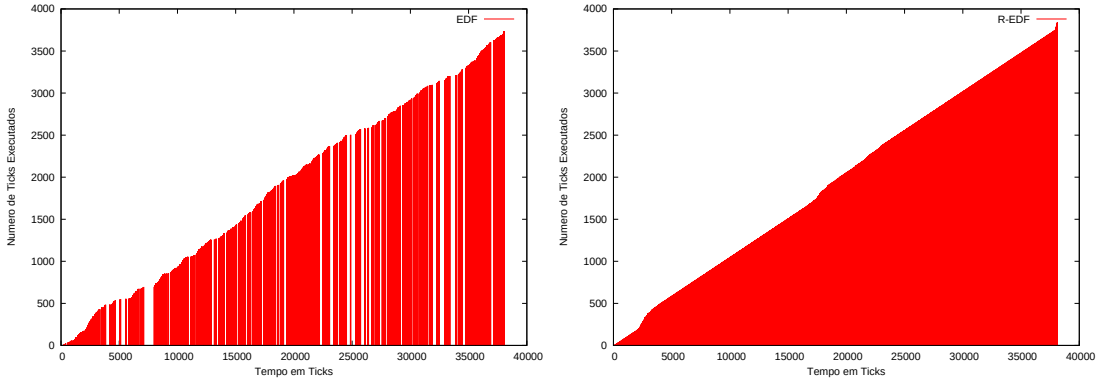
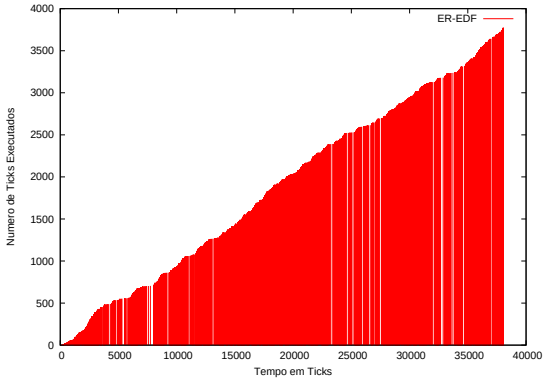


Figura 29: Comparativo execução de tarefas de melhor-esforço - Teste 5.



(a) EDF

(b) R-EDF



(c) ER-EDF

Figura 30: Comparativo ilustrando *starvation* como falhas nos gráficos - Teste 5.

5 Conclusão

Neste trabalho foi apresentado um novo algoritmo de escalonamento para a provisão de QoS em aplicações de tempo-real. Denominado *Enhanced* EDF, ele é baseado no algoritmo R-EDF, que apresenta restrições na execução de tarefas de execução variável. ER-EDF sanou esses problemas e adicionou o suporte a tarefas *hard real-time*, que são indispensáveis para aplicações que exigem grande precisão no cumprimento de seus requisitos.

Uma plataforma de *hardware* contendo o processador Plasma foi prototipada. Plasma é um processador *soft-core* que implementa um subconjunto de instruções MIPS. Ele contém os periféricos necessários para a implantação de um SO embarcado como, por exemplo, *timer* e comunicação serial.

EPOS, um sistema operacional embarcado, orientado a objetos, concebido para aplicações profundamente embarcadas, foi utilizado para a implementação do algoritmo ER-EDF. Este SO foi modificado para executar na plataforma de *hardware*. Para a execução de testes comparativos, foram implementados três algoritmos de escalonamento: EDF, R-EDF e ER-EDF.

Para a execução e análise de testes foram concebidas duas ferramentas: QoS Tester e QoS Analyser. A primeira é responsável pela comunicação com a plataforma de *hardware* e o SO EPOS para a geração de *benchmarks*, execução de tarefas e captura de dados. A última é responsável por mapear e analisar os dados gerados fornecendo meios para geração de dados estatísticos para posterior análise.

ER-EDF apresentou significativa melhora em relação ao seu antecessor R-EDF. A adição do suporte a tarefas *hard real-time* permite que desenvolvedores configurem a reserva para tarefas com tais restrições. Entretanto, a melhora de performance em tarefas de tempo-real deu-se em detrimento a tarefas de melhor-esforço. Mesmo assim, ER-EDF consegue significativa melhora para diminuir tempos de *starvation* em relação ao EDF.

5.1 Trabalhos Futuros

Como proposta para trabalhos futuros, pode-se citar o acréscimo de melhorias à ferramenta QoS Analyser para fornecer ao usuário meios gráficos de verificar o comportamento da execução de tarefas. Isto facilitaria a detecção de peculiaridades no escalonamento e possibilitaria o aprimoramento de algoritmos.

Outra proposta em relação as ferramentas de teste é a criação de um banco de algoritmos de escalonamento de tempo-real para a comparação através de testes automatizados. Este banco

poderia incluir algoritmos clássicos como, por exemplo, o *Rate Monotonic* (RM).

Um estudo mais aprofundado sobre o comportamento de tarefas quando escalonadas pelo ER-EDF poderia ser realizado para aprimorar o algoritmo e evitar perdas desnecessárias em tarefas *soft real-time* de tempo de execução variável. O estudo do funcionamento de outros algoritmos como, por exemplo, RM, também poderia auxiliar na criação de um novo algoritmo ainda mais eficiente.

Por fim, uma valiosa contribuição seria a implementação de uma aplicação real com requisitos equivalentes ao estado-da-arte atual. Além de uma real verificação da aplicabilidade dos algoritmos estudados nesse trabalho, propõe-se que a implementação seja monitorada para verificar o quanto a adição de suporte a configuração de QoS auxilia no processo de desenvolvimento. Assim, o aprimoramento do algoritmo poderia voltar-se a atender requisitos de *time-to-market* de desenvolvimento de produtos.

Referências

[1] A. T. Campbell, *A Quality of Service Architecture*. PhD thesis, Computing Department Lancaster University, UK, Jan. 1996.

[2] C. D. Gill, R. K. Cytron, and D. C. Schmidt, "Multiparadigm scheduling for distributed real-time embedded computing," In: *IEEE Proceedings Special Issue on Modeling and Design of Embedded Software*, Jan. 2003, pp. 183–197.

[3] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. San Francisco, California: CMPBooks, 2003, pp. 294.

[4] R. A. Bergamaschi and W. R. Lee, "Designing Systems-on-Chip Using Cores," In: *Design Automation Conference*, June 2000, pp. 420–425.

[5] R. A. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, W. R. Lee, F. White, and J.-M. Daveau, "Automating the Design of SoCs Using Cores," In: *IEEE Design and Test of Computers*, Oct. 2001, pp. 32–45.

[6] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, Dec. 2000, pp. 1523–1543.

[7] W. Wolf, "How many system architectures?," *Embedded Computing*, vol. 36, Mar. 2003, pp. 93–95.

[8] T. T. Ye, L. Benini, and G. De Micheli, "Packetized On-Chip Interconnect Communication Analysis for MPSoC," In: *Design, Automation and Test in Europe Conference and Exposition*, Mar. 2003, pp. 10344–10349.

[9] A. A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*. New York: Morgan Kaufmann, 2005, pp. 581.

[10] W. Wolf, "The Future of Multiprocessor Systems-on-Chips," In: *Proceedings of the 41th Design Automation Conference*, June 2004, pp. 681–685.

[11] R. Neugebauer and D. McAuley, "Energy Is Just Another Resource: Energy Accounting and Energy Pricing in the Nemesis OS," In: *8th IEEE Workshop on Hot Topics in Operating Systems*, May 2001, pp. 59–64.

[12] J. Blanquer, J. Bruno, E. Gabber, M. McShea, B. Ozden, A. Silberschatz, and A. Singh, "Resource Management for QoS in Eclipse/BSD," In: *Proceedings of the First FreeBSD Conference*, Oct. 1999.

[13] M. F. Moreno, C. S. Neto, A. T. d. A. Gomes, S. Colcher, and L. F. G. Soares, “QoSOS: An Adaptable Architecture for QoS Provisioning in Network Operating Systems,” *Revista Da Sociedade Brasileira de Telecomunicações*, vol. 18, Oct. 2003, pp. 118.

[14] M. Pastrnak, P. Poplavko, P. H. N. de With, and J. van Meerbergen, “Novel QoS Model for Mapping of MPEG-4 Coding onto MP-NoC,” In: *9th IEEE International Symposium on Consumer Electronics*, June 2005, pp. 93–98.

[15] L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm,” *IEEE Computer*, vol. 35, no. 1, 2002, pp. 70–78.

[16] S. Kumar, “On Packet Switched Networks for on-Chip Communication,” In: *Networks on Chip*, Hingham, MA, USA: Kluwer Academic Publishers, 2003, pp. 85–106.

[17] W. Yuan, K. Nahrstedt, and K. Kim, “R-EDF: A Reservation-Based EDF Scheduling Algorithm for Multiple Multimedia Task Classes.,” In: *IEEE Real Time Technology and Applications Symposium*, May 2001, pp. 149–154.

[18] I. Xilinx, *Spartan-3 Starter Kit Board - User Guide*, 2005. Capturado em: http://www.xilinx.com/bvdocs/userguides/ug130.pdf, Junho 2006.

[19] A. A. Fröhlich and W. Schröder-Preikschat, “High Performance Application-Oriented Operating Systems – the EPOS Aproach,” In: *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, Sept. 1999, pp. 3–9.

[20] A. A. Fröhlich, *Application-Oriented Operating Systems*. Berlin, TU: Sankt Augustin GMD Forschungszentrum Informationstechnik, 2001, pp. 210.

[21] L. Abeni and G. C. Buttazzo, “Resource Reservation in Dynamic Real-Time Systems,” *Real-Time Systems*, vol. 27, no. 2, 2004, pp. 123–167.