Modelagem e Implementação de Escalonadores de Tempo Real para Sistemas Embarcados

Hugo Marcondes[1], **Rafael Cancian**[2]
Marcelo Stemmer[2], **Antônio Augusto Fröhlich**[1]

[1]Laboratório de Integração de Software e Hardware
Universidade Federal de Santa Catarina
Caixa Postal 476 – 88049-900 – Florianópolis – SC – Brazil

[2]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina

{hugo,guto}@lisha.ufsc.br,{cancian,marcelo}@das.ufsc.br

Abstract. *Embedded systems frequently require an integrated hardware/-software design within real time constrains. In order to achieve such contrains, an adequate selection of a scheduling policy must be done. This work proposes the design and implementation of real time schedulers for embedded systems, within the context of Application Oriented System Design (AOSD). The use of AOSD enabled the development of schedulers where the policy is detached from the scheduling mechanism, fostering a better reusability of the scheduling components. The results shows that such design could be implemented to scale from 8 bits microcontrollers, 32 bits architectures and to specific hardware implemented design.*

Resumo. *Devido a suas características, sistemas embarcados freqüentemente demandam um projeto integrado de software e hardware com restrições de tempo real. Para que tais restrições sejam respeitadas, uma política de escalonamento de tarefas adequada deve ser selecionada. Este trabalho apresenta a modelagem e implementação de escalonadores de tempo real para sistemas embarcados, no contexto do projeto de sistemas orientados à aplicação. Esta abordagem permitiu a separação da política de escalonamento e seu mecanismo, promovendo uma maior reusabilidade dos artefatos envolvidos. Os resultados apresentados demonstram que esta implementação permite o seu uso em microcontroladores de 8 bits, arquiteturas de 32 bits, e até mesmo para implementações dedicadas de hardware.*

1. Introdução

Sistemas operacionais para sistemas dedicados devem ser adaptados para prover apenas o suporte necessário a uma aplicação bem definida. Desta forma, fatorar o sistema operacional em componentes selecionáveis e configuráveis é uma boa alternativa para modelar e projetar sistemas operacionais dedicados. Por outro lado, sistemas dedicados freqüentemente demandam um projeto integrado de software e hardware e apresentam uma grande variedade em termos de arquiteturas de hardware, desde microcontroladores de 8 bits até ao projeto de chips dedicados (ASIC), dificultando a tarefa de modelar e implementar

componentes reusáveis que possam ser efetivamente aplicados nessa gama de arquiteturas.

Dentre as principais famílias de componentes que formam um Sistema Operacional Embarcado (SOE) estão as tarefas, os temporizadores, os sincronizadores e os escalonadores de tarefas. Entretanto, essas famílias de componentes estão intrinsecamente relacionadas, de modo que, sem o devido projeto, um componente não pode ser modificado/adaptado sem influenciar os demais. No caso de sistemas de tempo real, tais componentes devem ainda ser projetados de forma a garantir restrições tais como a previsibilidade do tempo máximo de execução destes.

Mesmo numa única família, a adaptação de componentes para diferentes cenários de execução pode não ser fácil. Escalonadores de tarefas, por exemplo, apresentam uma miríade de algoritmos e características, incluindo escalonadores de tempo real altamente especializados e relativamente complexos. Nesses casos, permitir que um sistema operacional embarcado possua suporte a qualquer algoritmo (tempo real ou não), independente de suas características, sem exigir alterações no código de outras partes e componentes do sistema, é um grande desafio; principalmente em um sistema embarcado onde há grandes restrições de recursos computacionais.

Além das grandes diferenças algorítmicas e conceituais nos escalonadores, o projeto integrado de software e hardware desses sistemas permite que as funcionalidades tipicamente encontradas nos sistemas operacionais possam ser implementadas em hardware, seja através de dispositivos de lógica programável e até mesmo através do projeto de ASICs, sendo comum a implementação de escalonadores de tarefas em hardware, dado que esse é um componente executado com muita freqüência e fonte de sobrecusto (*overhead*). Assim, um sistema operacional embarcado adaptável à aplicação deve permitir que esse componente seja implementado em ambos os domínios (software e hardware) de forma eficiente.

Todos esses problemas não podem ser resolvidos simplesmente com uma implementação cuidadosa, e demandam um projeto de sistema adequado e engenhoso. Neste artigo, focamos na descrição da análise e modelagem dos principais componentes relacionados aos escalonadores e aspectos avançados de implementação que, apenas juntos, permitem a solução adequada dos problemas apresentados. Para a modelagem foi utilizada a metodologia de engenharia de domínio, segundo a AOSD (*Application Oriented System Design*) [Fröhlich 2001], que agrega uma série de paradigmas de programação para guiar o projeto de sistemas adaptados à aplicação. Nossas contribuições incluem um modelo eficiente e a apresentação de técnicas de implementação para adaptação de escalonadores em sistemas operacionais embarcados orientados à aplicação, além de permitir sua execução em diferentes arquiteturas, incluindo pequenos microcontroladores de 8 bits.

Este artigo está organizado da seguinte forma: A seção 2 apresenta conceitos e a descrição de alguns escalonadores encontrados na bibliografia, bem como a descrição das principais técnicas utilizadas neste trabalho. As seções 3 e 4 apresentam o desenvolvimento realizado e resultados alcançados, o que inclui o modelo conceitual básico e aspectos da implementação realizada. Por fim, a seção 5 apresenta algumas conclusões e considerações finais.

2. Fundamentação e Trabalhos Relacionados

O escalonamento de tarefas é considerado o coração de um sistema operacional, e dezenas de diferentes escalonadores têm sido propostos, principalmente escalonadores de tempo real para classes de aplicações específicas. Muitos podem ser implementados através da ordenação em uma fila de tarefas prontas, conforme um critério determinado. Isso inclui os escalonadores mais conhecidos, como FIFO, alternância circular (*round-robin*), prioridade, SPF (*Shortest Process First*), RM (*Rate Monotonic*), EDF (*Earliest Deadline First*) entre vários outros [Harvey DEITEL 2005]. Entretanto, outros algoritmos são muito mais complexos. Algoritmos como DSS (*dynamic sporadic server*) e *dynamic priority exchange server* [Buttazzo 1997] permitem o uso de filas separadas para tarefas periódicas e aperiódicas e também que pelo menos uma tarefa periódica especial atenda tarefas aperiódicas com regras específicas de temporização, consumo e concessão de "créditos" (*budgets*); Algoritmos como *Elastic Task Model* [G.C. Buttazzo and Abeni 1998] permitem mudar parâmetros das tarefas, como seu período, para adaptar-se à carga atual do sistema. Vários outros exemplos poderiam ilustrar as grandes diferenças entre as dezenas de escalonadores de tarefas propostos na literatura, de modo que suportá-los de forma transparente não é uma tarefa óbvia.

Por sua importância, vários sistemas operacionais embarcados já permitem a adaptação de seus escalonadores, mesmo dinamicamente. Entretanto, essa adaptação normalmente restringe-se a alguns poucos algoritmos específicos que alteram apenas a ordenação de uma fila, como FIFO, *round-robin*, prioridade, EDF e RM. Algumas poucas soluções permitem a subtituição por algoritmos mais elaborados. Entre elas está o S.Ha.R.K. (uma evolução do Hartik) [Shark 2008], que inclui algoritmos como PS (*Polling Server*), DS (*Deferrable Server*), SS (*Sporadic Server*), CBS (*Constant Bandwidth Server*) e CBS-FT (*Fault Tolerant CB*). Além disso, muitos sistemas operacionais de tempo real usam algoritmos de escalonamento que não consideram o prazo máximo de execução das tarefas (*deadline*), ou seja, usam escalonadores não tempo real para escalonar tarefas tempo real. Entretanto, além de exigir garantias de atendimento dos prazos das tarefas, muitas aplicações embarcadas de tempo real exigem ainda mais dos escalonadores. Dentre vários exemplos, podemos citar aplicações multimídia, que exigem que a taxa de execução das tarefas de vídeo e áudio seja constante (minimizando o *jitter*). Para atender esse tipo de exigência faz-se necessária a utilização de algoritmos específicos, como o CBS [G.C. Buttazzo and Abeni 1998], já que os algoritmos usuais não são adequados, pois desconsideram totalmente o *jitter*. Disso conclui-se que não há um suporte adequado a esse tipo de aplicação quando o SOE não provê escalonadores específicos, e esse é o caso de muitos SOE, inclusive de tempo real.

Grande parte dos sistemas operacionais de tempo real disponíveis hoje, tal como o *Embedded RT Linux*, QNX e VxWorks, tem seu uso prático em plataformas profundamente embarcadas limitado, devido ao tamanho do código gerado e a dificuldade de portabilidade. Além disso, a grande maioria dos sistemas operacionais de tempo real não consideram co-design em seu projeto e, portanto, ignoram as possibilidades de configuração do hardware. Assim, embora haja muitas alternativas supostamente disponíveis de sistemas operacionais de tempo real para sistemas embarcados, poucas realmente são aplicáveis a várias arquiteturas (principalmente aquelas com maiores restrições de recursos) e adaptáveis às necessidades das aplicações-alvo.

Vários trabalhos de hardware/software co-design para sistemas de tempo real já foram propostos. Implementações em hardware para escalonadores de tarefas foram propostos, dentre outros, por [Mooney and Micheli 2000], que implementaram um escalonador cíclico, e por [P. Kuacharoen and Mooney 2003], que implementaram os algoritmos de prioridade, RM e EDF. Além do suporte para escalonamento de tarefas, [Kohout and Jacob 2003] desenvolveram suporte para gerenciamento do tempo e de eventos, pois são atividades muito comuns aos STR e com alto paralelismo intrínseco. Entretanto, uma limitação desse suporte é que apenas escalonamentos com prioridade fixa são possíveis. O projeto *HThread* [Anderson et al. 2006] propõem um modelo de programação que permite que tarefas implementadas em hardware interajam com tarefas em software, através da implementação de escalonadores e dispositivos de sincronização em ambos os domínios (hardware e software). Outros tipos de suporte também foram propostos, como o de gerenciamento de memória [Shalan and Mooney 2000] e o de protocolos de acesso a recurso [Akgul 2003], que implementa o protocolo de herança de prioridade para evitar *deadlocks* e o bloqueio ilimitado de tarefas.

Nesse cenário, o EPOS (*Embedded Parallel Operating System*) surge como uma alternativa viável de sistema operacional de tempo real multiplataforma para sistemas embarcados. O EPOS compreende um *framework* e ferramentas para geração de sistemas operacionais, sendo um produto da *Application-Oriented System Design* (AOSD) [Fröhlich 2001], que combina diversos paradigmas de projeto que visam guiar o desenvolvimento de componentes de software altamente adaptaveis e reutilizáveis. A AOSD incluiu avanços como os adaptadores de cenários [Fröhlich and Schröeder-Preikschat 2000] e os mediadores de hardware [Polpeta and Fröhlich 2004] que permitem grande eficiência na geração automática de sistemas operacionais dedicados à aplicação. Posteriormente o EPOS foi expandido para gerar automaticamente não apenas o suporte de software, mas também o suporte de hardware (IPs - *Intelectual Properties*) necessários e suficientes para a aplicação, ou seja, a geração automática de SoCs (*Systems-on-a-chip*) orientadas à aplicação, já detalhado em publicações anteriores [Polpeta and Fröhlich 2005]. Atualmente EPOS tem portes funcionais para diversas arquiteturas entre elas, IA32, PPC, SparcV8, MIPS e AVR.

3. Análise e Modelagem

O processo de análise e modelagem de escalonadores de tarefas tempo real iniciou com a realização da engenharia deste domínio, de acordo com a AOSD, de forma a identificar as principais semelhanças e diferenças entre os conceitos do domínio, viabilizando desta forma a identificação das entidades que compõem o domínio de escalonamento de tempo real. A figura 1 apresenta o modelo conceitual de classes destas entidades.

Neste modelo conceitual, as tarefas são representadas através da classe `Thread`, que define o fluxo de execução da tarefa, implementando as funcionalidades tradicionais deste tipo de abstração encontrada na literatura. Esta classe também atende apenas aos requisitos de tarefas aperiódicas. A definicação de uma tarefa periódica é realizada através de uma especialização da classe `Thread` que agrega a esta mecanismos para a reexecução do fluxo de forma periódica, através do uso da abstração `Alarm`, responsável por reativar a tarefa sempre que um novo período se inicia. A classe `Alarm` por sua vez utiliza um `Timer` que irá fornecer o gerenciamento da passagem do tempo para o `Alarm`. Cada `PeriodicThread` possui seu próprio `Alarm` (embora alarmes possam

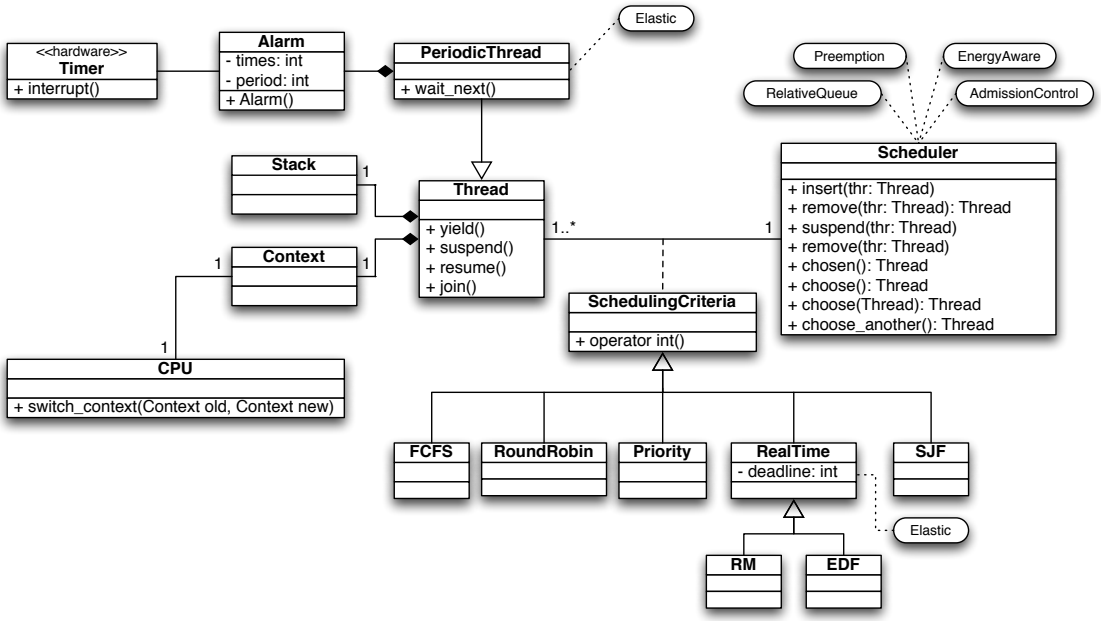


Figura 1. Modelo proposto para escalonadores de tarefas

existir sem threads periódicas), que podem compartilhar o(s) *timer(s)* da arquitetura. Cabe destacar que este é um modelo conceitual, e não um modelo de implementação. Assim, por exemplo, embora um `Alarm` conceitualmente utilize um `Timer` de hardware, essa associação exige a implementação de um mediador de hardware (*driver*) do `Timer`, que não aparece no modelo. Outras classes relacionadas também não foram apresentadas pois não estão diretamente associadas ao escalonador, foco deste trabalho.

As classes `Scheduler` e `SchedulingCriteria` definem a estrutura para realizar o escalonamento das tarefas. Ressalta-se umas das principais diferenças entre as abordagem tradicionais de modelagem de escalonadores, que geralmente apresentam uma hierárquia de especializações de um escalonador genérico, de forma a extende-lo para outras políticas de escalonamento. De forma a reduzir a complexidade de manutenção do código, geralmente ocasionada pelo uso de uma hierárquia complexa de especializações, assim como promover o reuso de código, o modelo separa a sua política de escalonamento (*scheduling criteria*) de seu mecanismo (implementação de filas). Esta separação é uma decorrência do processo de engenharia de domínio que permitiu identificar os aspectos comuns a todas as políticas de escalonamento, permitindo a separação destes aspectos (contidos no componente `Scheduler`) da caracterização de tais políticas (suas diferenças, expressas no componente `SchedulingCriteria`).

Esta separação entre o mecanismo e a política de escalonamento foi fundamental para a implementação de escalonadores em hardware. De fato, o escalonador em hardware implementa apenas o mecanismo de escalonamento, que realiza a ordenação das tarefas baseado na política selecionado. Isto permite que o mesmo componente em hardware possa ser utilizado independente da política selecionada. A separação do mecanismo de escalonamento e a sua política é realizada através da separação do algoritmo de ordenamento e o mecanismo de comparação entre os elementos que compõem a lista

do escalonador. Desta forma, cada política de escalonamento define a maneira como os elementos serão ordenados na respectiva fila.

Adicionalmente, durante o processo de análise e de engenharia de domínio foi identificada uma série de características que, segundo a AOSD, definem propriedades configuráveis (`configurable features`) de seus componentes [Fröhlich 2001]. De fato, tais características representam pequenas variações de uma entidade do domínio que podem ser ativadas ou não, alterando de forma sutil o comportamento do mesmo. Dentre tais propriedades configuráveis, foi identificada a característica do escalonamento ser preemptivo, o controle de admissão das tarefas, assim como a consideração de parâmetros de consumo de energia, permitindo que o mecanismo realize também políticas de qualidade de serviço (QoS) [Wiedenhoft and Fröhlich 2008].

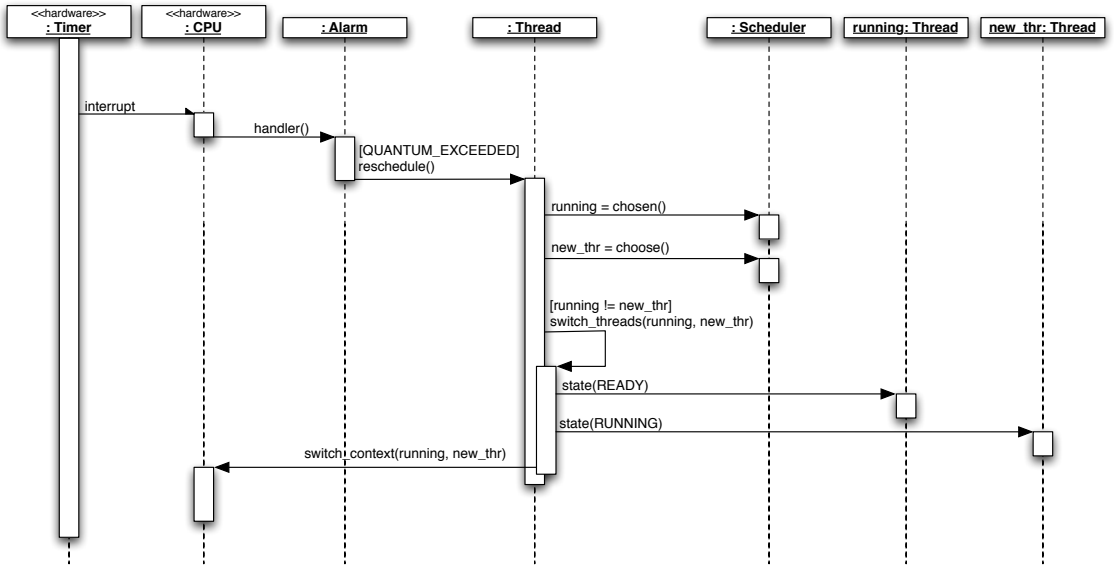


Figura 2. Diagrama de seqüência do reescalonamento de tarefas.

Outra característica identificada é relativa aos escalonadores que precisam alterar propriedades do modelo de tarefas utilizado. Conforme visto na seção de fundamentação, algoritmos de escalonamento elástico (como o *elastic task model*), permitem que o período das tarefas periódicas possam ser aumentados, caso a taxa de utilização da CPU esteja alta, e depois restaurados. Outros escalonadores não triviais, como o CBS e DSS possuem característica análoga. Essa característica é modelada como uma propriedade configurável aplicável aos `SchedulingCriteria` relativos a tarefas periódicas, assim como a classe `PeriodicThread`, habilitando funções que alteram a periodicidade ou outra propriedade das tarefas, uma vez que a política de escalonamento a solicite. Desta forma, mesmo algoritmos complexos podem ser suportados e adaptados sem exigir especializações de classes que complicariam o projeto.

De forma a ilustrar as interações entre os componentes do escalonador proposto, a figura 2 apresenta a interação dos componentes durate o rescalonamento ocorrido quando a fatia de tempo concedida para a tarefa em execução termina. Neste contexto, o `Timer` é responsável por gerar interrupções periódicas, que são contadas pelo `Alarm`. Quando o estouro da fatia de tempo concedido a tarefa atual é expirado, o `Alarm` invoca o método

da classe `Thread` para solicitar o reescalonamento das tarefas. Este por sua vez irá verificar qual é a tarefa que está em atual execução, assim como invocar o método *choose()* do `Scheduler`. Este retorna um ponteiro para a tarefa que deve ser executada. Neste ponto é realizada uma verificação para identificar se é necessário realizar uma troca de contexto de execução para uma tarefa de maior prioridade. Caso a troca seja necessária, os estados das tarefas envolvidas no processo são atualizados e uma troca de contexto é realizada na `CPU`; caso contrário o processo finaliza, mantendo a tarefa atual em execução.

Novamente, ressalta-se que a figura 2 representa um modelo conceitual, e não de implementação. As classes `Timer` e `CPU` apresentadas representam hardware, embora possuam implementação de seus respectivos mediadores em software. As mensagens `interrupt` e `handler`, deste modo, não representam métodos implementados em software, mas sim o sinal elétrico de interrupção e a invocação, pelo hardware, do tratador dessa interrupção, respectivamente. A mensagem `switch-context`, por sua vez, corresponde a um método implementado no mediador de hardware da CPU.

4. Implementação e Resultados

Esta seção apresenta os detalhes de implementação dos principais componentes do escalonador proposto, em especial a implementação do mecanismo de escalonamento em software e em hardware. Em seguida é apresentado como as principais políticas de escalonamento baseado foram implementadas através das `SchedulingCriteria`.

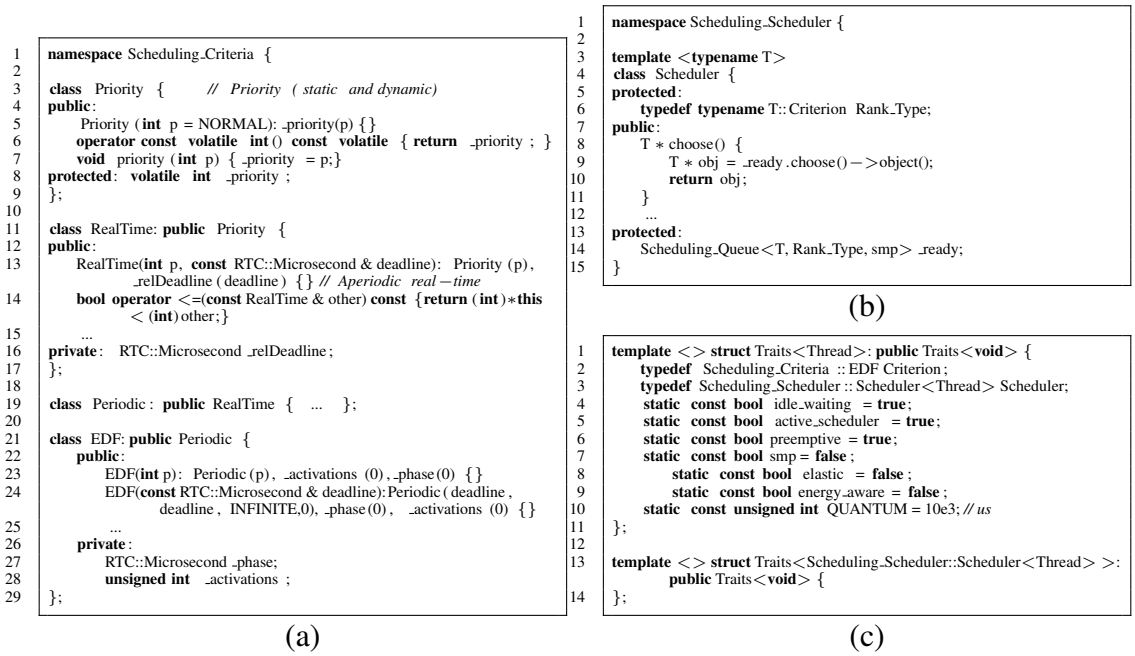
4.1. Escalonador em software

A implementação do escalonador em software segue o modelo tradicional de lista. Esta lista pode ser configurada para ser implementada utilizando uma ordenação convencional de seus elementos, assim como uma ordenação de forma relativa, onde cada elemento armazena o seu parâmetro de ordenamento pela diferença deste com o elemento anterior, ou seja, seu parâmetro será sempre relativo a seu predecessor e assim por diante. Esta estrutura se torna especialmente interessante na implementação de políticas que possuem o seu ordenamento influenciado pela passagem do tempo, como o algoritmo EDF. Uma vez que o tempo é sempre crescente (como o *deadline* absoluto, critério utilizado pelo EDF), a utilização de uma lista convencional resultará em um estouro do limite das variáveis após um certo tempo. Esse tempo pode ser de apenas algumas horas, dependendo da freqüência, em um microcontrolador de 8 bits, causando o comportamento incorreto e inesperado do escalonador, o que é inadmissível em um sistema de tempo real. O uso de uma lista relativa nesses casos, elimina o problema de estouro de variável, cuja ocorrência é dificilmente detectada.

Independente do uso de filas relativas ou convencionais, o critério utilizado pelo algoritmo de ordenamento da fila é realizado pelo `SchedulingCriteria`. De forma geral, este componente pode ser visualizado como uma especialização do tipo inteiro que define o ordenamento da fila, e implementa a sobrecarga de seus operadores aritméticos e de comparação, de forma a estabelecer políticas mais complexas de ordenamento, exigidas por vários algoritmos. Por exemplo, no caso de algoritmos multifilas, a `SchedulingCriteria` pode encapsular dois parâmetros de ordenamento: a identificação da fila e a prioridade do elemento dentro desta fila, além de sobrecarregar o operador de comparação menor/igual ($\leq$) para que ambos os parâmetros sejam avaliados durante a comparação entre dois elementos, durante a sua inserção ordenada no

mecanismo de escalonamento. O uso de sobrecarga de operadores mantém o projeto elegante, evita a grande hierarquia de classes e provê suporte adequado a algoritmos mais complexos.

A figura 3(a) apresenta alguns trechos da implementação dos critérios de escalonamento, em que pode-se observar a sobrecarga do operador `int()` (linha 6) e do operador $\leq$ (linha 14). A figura 3(b) apresenta parte do escalonador metaprogramado, tornando-se independente da entidade `T` a ser escalonada (normalmente `Thread`) e da implementação da lista (também metaprogramada). A figura 3(c) apresenta parte do arquivo de configuração do SOE em que é possível especificar a política e o mecanismos a serem utilizados.



```
1  namespace Scheduling_Criteria {
2
3  class Priority {        // Priority ( static and dynamic)
4  public:
5       Priority (int p = NORMAL): _priority(p) {}
6       operator const volatile int() const volatile { return _priority ; }
7       void priority (int p) { _priority = p;}
8  protected: volatile int _priority ;
9  };
10
11 class RealTime: public Priority {
12 public:
13      RealTime(int p, const RTC::Microsecond & deadline): Priority (p),
                _relDeadline ( deadline ) {} // Aperiodic real−time
14      bool operator <=(const RealTime & other) const {return (int)*this
                < (int) other;}
15        ...
16 private: RTC::Microsecond _relDeadline;
17 };
18
19 class Periodic : public RealTime { ... };
20
21 class EDF: public Periodic {
22      public:
23           EDF(int p): Periodic (p), _activations (0), _phase(0) {}
24           EDF(const RTC::Microsecond & deadline):Periodic ( deadline ,
                   deadline , INFINITE,0), _phase(0), _activations (0) {}
25            ...
26      private:
27           RTC::Microsecond _phase;
28           unsigned int _activations ;
29 };
```

(a)

```
1  namespace Scheduling_Scheduler {
2
3  template <typename T>
4  class Scheduler {
5  protected:
6       typedef typename T::Criterion Rank_Type;
7  public:
8       T * choose() {
9            T * obj = _ready.choose()->object();
10           return obj;
11      }
12       ...
13 protected:
14      Scheduling_Queue<T, Rank_Type, smp> _ready;
15 }
```

(b)

```
1  template <> struct Traits<Thread>: public Traits<void> {
2       typedef Scheduling_Criteria ::EDF Criterion;
3       typedef Scheduling_Scheduler :: Scheduler<Thread> Scheduler;
4       static const bool idle_waiting = true;
5       static const bool active_scheduler = true;
6       static const bool preemptive = true;
7       static const bool smp = false;
8            static const bool elastic = false;
9            static const bool energy_aware = false;
10      static const unsigned int QUANTUM = 10e3; // us
11 };
12
13 template <> struct Traits<Scheduling_Scheduler::Scheduler<Thread> >:
            public Traits<void> {
14 };
```

(c)

Figura 3. Implementação dos critérios de escalonamento (a); escalonador (b); seleção do algoritmo utilizado (c)

4.2. Escalonador em hardware

O componente `Scheduler` foi implementado como um componente híbrido de hardware e software [Marcondes and Fröhlich 2008], e desta forma, uma implementação em hardware deste componente também foi realizada. A figura 4 apresenta a organização de blocos lógicos da implementação do componentes em hardware.

Basicamente, este componente implementa uma lista ordenada de elementos que é armazenada em uma memória interna do componente. Um módulo controlador (`Controladora`) é responsável por interpretar os dados recebidos pela interface do componente em hardware e invocar o processo correspondente com a funcionalidade requisitada (através do sinal de `command`, da interface). Esta implementação, tal qual a implementação em software realiza a inserção de elementos na fila de escalonamento de forma ordenada, ou seja, a fila é sempre mantida ordenada, de acordo com as informações contidas no `SchedulingCriteria`. Internamente a este componente, uma lista-ligada duplamente encadeada é implementada.

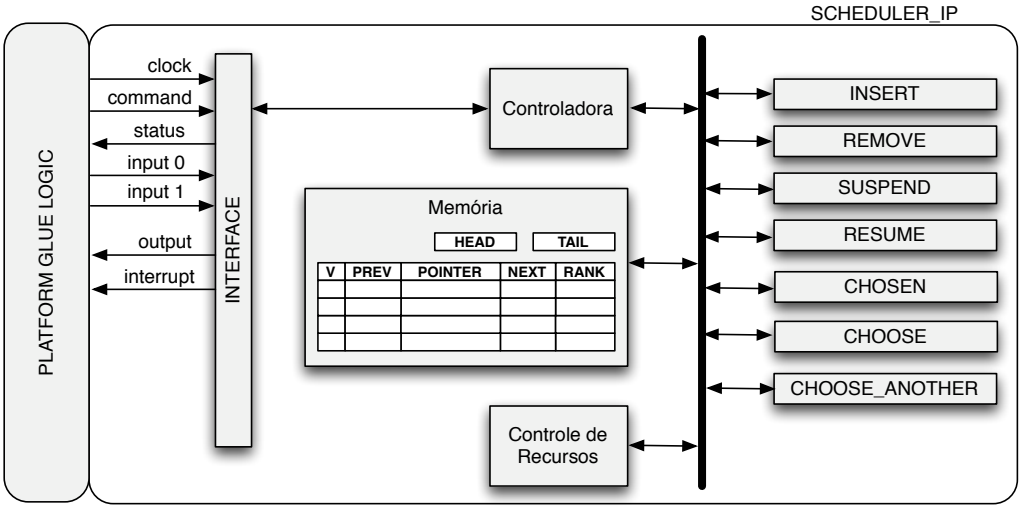


Figura 4. Diagrama de blocos do componente escalonador em hardware.

É importante ressaltar dois aspectos da implementação deste componente devido a restrições inerentes de sua implementação em hardware, em especifico com o foco em dispositivos de lógica programável.

Cabe destacar dois aspectos da implementação deste componente, que foram considerados devido às restrições existentes em dispositivos de lógica programável, os quais foram utilizados para prototipação deste componente. Tais aspectos estão relacionados com a restrição de recursos provenientes destes dispositivos. Idealmente um escalonador em hardware deveria explorar ao máximo o paralelismo inerente deste, contudo, o custo pela exploração máxima deste paralelismo em termos de consumo de blocos lógicos da FPGA se torna extremamente alto, principalmente na implementação da comparadores paralelos para realizar a busca de elementos e também a procura pela posição de inserção de elementos.

Em especial, o uso de ponteiros de 32 bits, para expressar referências aos elementos armazenados na lista (neste caso `Threads`) torna-se muito custoso, quando se deseja realizar uma pesquisa por esses ponteiros, e consequentemente utilizar comparadores de 32-bits. Por outro lado, o número máximo de `Threads` a serem escalonadas em um sistema embarcado é conhecido de antemão, e por isso, foi adotado um mapeamento entre o endereçamento de objetos da arquitetura (do tamanho da palavra da arquitetura) para um endereçamento interno a este componente que irá utilizar tantos bits quanto forem necessários para endereçar apenas o número máximo de elementos que este componente suporta, reduzindo assim, a lógica gasta pela implementação do mesmo.

O outro aspecto está relacionado à busca pela posição do elemento durante a inserção na fila. Idealmente, a busca por esta posição poderia ser implementada através de uma comparação paralela de todos os elementos da lista, de forma a encontrar a posição de inserção em apenas um ciclo de execução. Contudo, esta abordagem, além de aumentar o consumo de recursos e lógica conforme já mencionado, ainda gera um impacto no atraso máximo do circuito devido à sua maior complexidade, reduzindo também a freqüência de operação do mesmo. Desta forma, se optou pela implementação de uma

busca seqüencial pela posição de inserção na lista, percorrendo a mesma. Nesta abordagem, apesar do tempo de inserção de elementos variar, essa variação pode ser, na maioria dos casos, escondida pelo paralelismo entre a execução do componente em hardware e a CPU. Desta forma, durante a operação de inserção, o controle é imediatamente devolvido a CPU, enquanto o hardware executa a inserção do elemento na fila.

4.3. Avaliação da implementação do modelo proposto

A avaliação da implementação do modelo proposto foi realizada através de uma aplicação sintética de tempo real, onde foi definido um conjunto de tarefas periódicas hipotéticas. A configuração dos componentes do EPOS foi realizada através das ferramentas desenvolvidas para o sistema que gera um conjunto de parâmetros que efetuam a ligação da interface utilizada pela aplicação à respectiva implementação do componente selecionado [Figura 3(c)]. A figura 5 ilustra a implementação desta aplicação teste. A figura 5 (a) apresenta a implementação de cada tarefa, sendo que neste caso, a tarefa consome ciclos de CPU, simulando sua ocupação da CPU. A figura 5 (b) apresenta a aplicação principal, que é responsável por ativar e criar no sistema as tarefas que serão executadas (linhas 5–7), definindo os parâmetros de de cada tarefa, de acordo com a política de escalonamento utilizada, neste caso, é utilizado a política EDF, definindo desta forma o período, deadline e número de ativações que a tarefa deve ter.

```
1  int Tn(int n, RTC::Microsecond wcet) {
2      RTC::Microsecond now, miliexec;
3      int activations = 0;
4
5      while (1) {
6          now = Alarm::elapsed();
7
8          // waste time in CPU
9          miliexec = 0;
10         do {
11             while (now == Alarm::elapsed());
12             miliexec++;
13             now = Alarm::elapsed();
14         } while (miliexec < wcet);
15
16         activations ++;
17         Periodic_Thread :: wait_next();
18     }
19     return 0;
20 }
```

(a)

```
1  int main() {
2      cout << "Main will create the periodic threads ...\n";
3
4      Periodic_Thread *t1, *t2, *t3;
5      t3 = new PeriodicThread(&Tn, 3, 60e3, SchedulingCriteria (300e3,300e3,10));
6      t2 = new PeriodicThread(&Tn, 2, 40e3, SchedulingCriteria (200e3,200e3,10));
7      t1 = new PeriodicThread(&Tn, 1, 20e3, SchedulingCriteria (100e3,100e3,10));
8
9      cout << "Main will wait for periodic threads to finish ...\n";
10     int status1 = t1->join();
11     int status2 = t2->join();
12     int status3 = t3->join();
13
14     cout << "Main will destroy periodic threads ...\n";
15     delete t1; delete t2; delete t3;
16
17     cout << "Main will finish ...\n";
18     return 0;
19 }
```

(b)

Figura 5. Aplicação de teste: (a) código de cada tarefa e (b) criação das tarefas

Esta aplicação foi compilada para as arquiteturas PowerPC (32 bits) e AVR (8 bits), utilizando os algortimos de escalonamento EDF, RATE MONOTONIC e PRIORIDADE. A tabela 1 apresenta o tamanho da aplicação gerada para cada política e arquitetura testada. Os testes também demonstraram o correto funcionamento das políticas utilizadas.

	PPC32		AVR8	
	.text	.data	.text	.data
EDF	51052	300	49246	853
Rate Monotonic	47908	272	36800	1003
Prioridade	47864	272	36790	1003

Tabela 1. Memória consumida pela aplicação teste

Os testes também foram realizados utilizando o escalonamento em hardware. Neste caso, utilizando uma plataforma de experimentação da FPGA VIRTEX4, que integra um processador PowerPC 405 e blocos de lógica programável, permitindo assim a prototipação rápida de aceleradores em hardware dedicados.

# Máx. Tarefas	Logic Usage	Slices	Máx. Freq.
2	5%	326	214.6 Mhz
4	10%	551	161.5 Mhz
8	19%	1078	138.8 Mhz
16	36%	2015	123.4 Mhz
24	51%	2833	114.6 Mhz
32	73%	3997	113.4 Mhz
48	103%	5665	82.0 Mhz

Tabela 2. Resultados de utilização da FPGA para o componente `Scheduler`

O modelo de FPGA disponível nesta plataforma de experimentação (ML 403) é o modelo `XC4VFX12` que provê 5,412 slices de lógica para a implementação dos aceleradores. A tabela 2 apresenta a área consumida desta FPGA, de acordo com o número máximo de tarefas que o escalonador pode gerenciar.

5. Conclusões

Este artigo apresentou a modelagem para a implementação de escalonadores de tarefas tempo real, de acordo com a AOSD. O uso de técnicas como a engenharia de domínio possibilitou o isolamento das diferenças presentes nas políticas de escalonamento, possibilitando um melhor reúso dos artefatos de projeto (implementação das políticas de escalonamento e do mecanismo de escalonamento), e permitiu que a abordagem proposta seja utilizada não apenas no desenvolvimento de sistemas reais, assim como uma plataforma de experimentação de algoritmos de escalonamento de tempo real, viabilizando a execução de testes de políticas existentes, ou mesmo novas políticas em todas as arquiteturas suportadas pelo sistema EPOS, que variam desde microcontroladores de 8 bits a arquiteturas de 32 bits.

Referências

Akgul, B. (2003). "Hardware support for priority inheritance." In Publishers, K. A., editor, *24th IEEE International Real-Time Systems Symposium*.

Anderson, E., Agron, J., Peck, W., Stevens, J., Baijot, F., Komp, E., Sass, R., and Andrews, D. (2006). "Enabling a uniform programming model across the software/hardware boundary". In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 89–98.

Buttazzo, G. (1997). *Hard Real-Time Computing Systems*. Kluwer Academic Publishers.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. PhD thesis, Sankt Augustin: GMD - Forschungszentrum Informationstechnik.

Fröhlich, A. A. and Schröeder-Preikschat, W. (2000). "Scenario adapters: Efficiently adapting components." In *Proceedings of 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA.

G.C. Buttazzo, G. L. and Abeni, L. (1998). “Elastic task model for adaptive rate control.” In *19th IEEE Real-Time Systems Symposium*, pages 286–295, Madrid, Spain.

Harvey DEITEL, Paul DEITEL, D. R. C. (2005). *Sistemas operacionais*. Prentice Hall.

Kohout, P. and Jacob, B. (2003). “Hardware support for real-time operating systems.” In *Proceedings of CODES - ISSS’03*, Newport Beach, CA - USA.

Marcondes, H. and Fröhlich, A. A. M. (2008). “On hybrid hw/sw components for embedded system design.” In *Proceedings of the 17th IFAC World Congress, 2008*, volume 17, Seoul, South Korea. IFAC.

Mooney, V. and Micheli, G. D. (2000). “Hardware/software codesign of run-time schedulers for real-time systems.” In *Proceedings of Design Automation of Embedded Systems*, pages 89–144.

P. Kuacharoen, M. S. and Mooney, V. (2003). “A configurable hardware scheduler for real-time systems.” In *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA’03*.

Polpeta, F. V. and Fröhlich, A. A. (2004). “Hardware mediators: a portability artifact for component-based systems.” In *Proceedings of International Conference on Embedded and Ubiquitous Computing*, volume 3207, pages 271–280.

Polpeta, F. V. and Fröhlich, A. A. (2005). “On the automatic generation of soc-based embedded systems.” In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*.

Shalan, M. and Mooney, V. (2000). “A dynamic memory management unit for embedded real-time system-on-a-chip.” In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 180–186.

Shark (2008). S.ha.r.k.: Soft hard real-time kernel. World Wide Web.

Wiedenhoft, G. R. and Fröhlich, A. A. (2008). “Gerência de energia no epos utilizando técnicas da computação imprecisa.” In *Proceedings of the Fifth Brazilian Workshop on Operating Systems*, pages 34–45.