Interferência das *Hard irqs* e *Softirqs* em Tarefas com Prioridade de Tempo Real no Linux

Andreu Carminati[1], **Rômulo Silva de Oliveira**[1]

[1]Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040–900 – Florianópolis – SC – Brasil

{andreu,romulo}@das.ufsc.br

Abstract. *The operating system Linux is nowadays an attractive alternative for a great spectrum of applications. Mainline Linux can be used for soft real-time. There are Linux variations (patch's and extensions) to further improve its capability of supporting soft real-time applications. However, some companies that use Linux embedded in products that need soft real-time prefer to use mainline Linux, since its maintenance and evolution is more guaranteed than those based on alternative patches. This paper is specifically about the interference generated by the executions of Softirqs and Hard irqs on tasks with maximum real-time priority in mainline Linux. Softirqs are those activities that happen inside of the kernel due to a hardware interrupt, but that are not executed immediately by the interrupt handler. The goal of this work is evaluate the impact of the Softirq and Hard irq activities on tasks with real-time priority.*

Resumo. *O sistema operacional Linux é atualmente uma alternativa atraente para um grande espectro de aplicações. O Linux mainline pode se usado para soft real-time. Existem variações do Linux (extensões e patches) que melhoram a sua capacidade em suportar aplicações do tipo soft real-time. Entretanto, muitas companhias que usam Linux embarcado em seus produtos que exigem características soft real-time preferem utilizar o mainline Linux, pois sua manutenção e evolução é mais garantida que as baseadas em patches alternativos. Este artigo é especificamente sobre a interferência gerada pela execução de Softirqs e Hard irqs em tarefas com prioridade máxima de tempo real no Linux mainline. Softirqs são aquelas atividades que acontecem dentro do kernel devido a interrupção de hardware, mas que não são executadas imediatamente pelo tratador de interrupção. O objetivo deste trabalho é avaliar o impacto das atividades das Softirqs e Hard irqs em tarefas de tempo real com prioridade máxima.*

1. Introdução

Com a crescente popularização das arquiteturas de 32 bits, torna-se viável a utilização do Linux como plataforma para aplicações de tempo real com diversos focos (aplicações embarcadas por exemplo). Isto ocorre devido a todas as vantagens que este sistema moderno de propósito geral pode oferecer, como ambiente multitarefa, pilha de rede, recursos gráficos, amplo suporte a praticamente todo tipo de *hardware*, estabilidade de código, evolução contínua e constantes atualizações para eliminação de falhas. Outra grande vantagem na utilização do Linux é a possibilidade de se poder estudar, alterar e fazer qualquer tipo

de ajuste que se faça necessário para adequá-lo a uma determinada aplicação ou classe de aplicações.

O grande foco de desenvolvimento do Linux é desempenho, estabilidade, escalabilidade e baixo tempo médio de resposta, e isso é fácil de se notar quando vemos o esforço que a comunidade de desenvolvedores está empreendendo no sentido de reforçar essas características. Vemos isso, por exemplo nas profundas mudanças introduzidas no escalonador a partir da versão 2.6 do kernel, que permitiu inclusive, o escalonamento com complexidade constante O(1) [Love 2005] e que já foi substituído pelo CFS[Molnar 2007] que corrige várias deficiências deste, e a introdução do conceito de domínios e grupos de escalonamento que começa a preparar o Linux para executar de modo eficiente nas arquiteturas NUMA (*Non-uniform Memory Access*).

Apesar de todos esses esforços, o requisito "tempo real" não é prioritário no desenvolvimento, mas isso não implica que o Linux relegue o desenvolvimento de suporte a tempo real. Muito pelo contrário, o Linux já possui nativamente implementado o padrão POSIX.4[POSIX.13 1998], o que define por exemplo, políticas de escalonamento de tempo real (vide SCHED_FIFO e SCHED_RR) que já suportam uma vasta gama de aplicações.

Existem muitos estudos que investigam a capacidade que o kernel do Linux tem em suportar aplicações com *deadlines* críticos, como os que investigaram as latências de tratamento de interrupções e troca de contexto de processos [Regnier et al. 2008]. O foco deste estudo é outro: estudo empírico das interferências em processos de tempo real com prioridade máxima. Em princípio, o estudo se focará no kernel padrão, ou seja, sem nenhuma alteração em nível estrutural que facilite e melhore a execução de tarefas em tempo real, como *patches* ou extensões. O objetivo é investigar até que ponto o Linux sozinho pode ser confiável no cumprimento dos *deadlines*.

Em um segundo momento, será abordado o uso do patch PREEMPT-RT, que é um patch não intrusivo no sentido de não utilizar nenhum tipo de camada de abstração de *hardware* (ao contrário de soluções como XENOMAI[Gerum 2004] e RTAI[Dozio and Mantegazza 2003]) e usa a mesma API POSIX presente no Linux padrão para execução de tarefas de tempo real, sendo simples a forma de comparação de características de execução de tarefas entre as duas versões do Linux. Devido a utilização da mesma API, não há necessidade de modificação da aplicação de teste.

2. Descrição do Problema

No Linux, tarefas com prioridade de tempo real estão sujeitas a diversos tipos de interferência durante o seu tempo de execução. O tipo mais comum de interferência está associado aos tratadores de interrupção de *hardware*, que são acionados de forma assíncrona à execução da tarefa. Esse tipo de interferência existe devido ao fato do Linux ser um sistema do tipo *Time Sharing*, onde essa abordagem melhora a reatividade do sistema a eventos externos, mas pode causar atrasos nos processos em execução, pois estes acumularão atrasos a cada novo atendimento. Outro tipo de interferência advém do uso de memória virtual, já que esta não garante presença imediata de um dado requisitado por um processo em memória primária, gerando assim possíveis atrasos devidos a operações de E/S. Isso pode se atenuado com o bloqueio de páginas em memoria primária, com o

uso de *mlockall* e *munlockall* que bloqueiam e desbloqueiam páginas, respectivamente, e também são definidas em POSIX[POSIX.13 1998]. Outro tipo de interferência observado com o uso de memória virtual vem do uso de TLB's (*translation lookaside buffer*) que auxilia a MMU na tradução rápida de endereços, quando *frames* de memória atualmente referenciados, já foram solicitados anteriormente. Caso contrário, ocorre um TLB *miss*. Outras interferências podem ocorrer, como as *Softirqs*, executadas, na maioria dos casos, na saída dos tratadores de interrupção.

A interferência causada por *Softirqs*, pode ser mais baixa que a gerada por tratadores de interrupção em cenários de baixa carga, pois o sistema terá pouca carga a ser postergada (em alguns casos, só processamento de *timers*). Entretanto, esta interferência cresce a medida que o sistema se vê forçado a processar ainda mais interrupções, como as de disco e rede (e todas aquelas que utilizam os mecanismos de adiamento de trabalho do *kernel*). Neste cenário *Softirqs* podem causar muito mais atrasos que tratadores de interrupção. Este estudo se focará basicamente na verificação da interferência causada pelas *Softirqs* e tratadores de interrupção (*Hard irqs*).

3. Descrição do Mecanismo de Tratamento de Interrupções, dos BH e Mecanismos de Postergação de Trabalho

O tratamento de interrupções no Linux em geral é dividido em duas partes: *Top Half* e *Bottom Half*[Love 2005]. A parte denominada *Top Half* está relacionada ao tratador de interrupção em si, que é executado assincronamente em resposta imediata (em tese, pois aqui ainda existem latências) à requisição de *hardware*. Esta parte precisa ser rápida, pois será executada com a atual linha de interrupção desabilitada (melhor caso), ou com todas as linhas do processador local desabilitadas (pior caso). Outro motivo importante é que tratadores lidam com *hardware* onde, em geral, as temporizações que regem a comunicação precisam ser muito precisas, daí são impostas mais restrições temporais. Nem todo tipo de trabalho pode ser realizado em um tratador de interrupção, pois ele não executa em contexto de processo, ou seja, não pode executar operações que levem a estados de bloqueio ou que possam colocar o tratador em *sleep*.

Feito o processamento básico (*handshake* e cópia de dados do dispositivo), seria interessante, de alguma forma, liberar o *hardware*, a linha atual, ou todas as linhas locais de interrupção (dependendo do caso) para novas interrupções e postergar o tratamento dos dados obtidos do dispositivo pelo tratador (se possível) para um momento mais oportuno. Dessa forma, foi criado o conceito de *Bottom Halves*, com o simples intuito de deferir para o futuro parte do trabalho de tratamento de interrupções. O uso de *Botton Halves* ajuda a manter o desempenho alto e o tempo de resposta baixo, pois é minimizado o tempo das interrupções, ou seja, o tempo gasto pelos tratadores de interrupção. Quando este trabalho é executado de certa forma não importa, desde que não seja executado no tratador em si (no *Top Half*), pelos motivos descritos acima.

3.1. Softirqs

São alocadas estaticamente, ou seja, em tempo de compilação. Está definida em: "linux/interrupt.h>". Atualmente, o *kernel* pode suportar no máximo 32 *Softirqs* (um array de 32 posições declarado em "kernel/softirq.c"). A execução das *Softirqs* ocorre (quando

marcadas para execução) após o processamento dos tratadores de interrupção, na *thread ksoftirqd*, ou em qualquer código que as execute explicitamente, através das chamadas à função do_softirq(). Um detalhe importante é que as *Softirqs* também executam em contexto de processo (através da *thread ksoftirqd*), mas em comparação com a execução na saída dos tratadores de interrupções, não representa uma parcela muito significativa.

3.2. Tasklets

São construídas sobre as *Softirqs*, ou seja, existe duas entradas no array das *Softirqs* (TASKLET_SOFTIRQ e HI_SOFTIRQ) que representam o conjunto das *Tasklets*. Apesar do nome, elas não tem nada a ver com as *Tasks* de *kernel* e, ao contrário das *Softirqs*, as *Tasklets* podem ser alocadas dinamicamente. Na maioria dos casos, é aconselhável utilizar *Tasklets* em vez de *Softirqs*, pois são mais simples de se utilizar, e possuem sincronização implícita, ou seja, a mesma *Tasklet* não irá executar em duas CPUs simultaneamente (comportamento que nas *Softirqs*, se necessário, deve ser explicitado com o uso de *spin-locks*).

3.3. Workqueues

Trata-se de um mecanismo de postergação de trabalho bem diferente das *Softirqs* e das *Takslets*, pois é executado por *threads de kernel*, ou seja, sempre em contexto de processo. São escalonáveis e podem executar operações que podem acarretar *sleep* (pode-se usar *mutexes* ao invés de *spin-locks*, acessar espaço de endereçamento de processos de usuário, fazer alocação de memória não atômica). O trabalho em si é executado pelo que é chamado de *worker threads*.

3.4. Threads de Kernel

Ainda é possível, como alternativa, criar *threads* específicas para processar trabalhos pendentes, mas é desaconselhável, já que existem os *BH's* e *workqueues*. O Processo é simples: Cria-se uma *thread* que é ativada apenas quando existe trabalho disponível.

3.5. Comentários

*Workqueue*s interferem somente em processos com os quais competem diretamente pelo uso do processador (mesma prioridade ou prioridade menor), pois as *worker threads* são gerenciadas pelo próprio escalonador do Linux, que irá evitar sempre que possível, inversões de prioridade. Apenas os tratadores de interrupção, *Softirqs* (consequência direta de serem executadas na saída dos tratadores de interrupção, ou seja, também em contexto de interrupção) e *Tasklets* (construídas em cima das *Softirqs*) causarão interferências significativas em processos com prioridade de tempo real (maximizada pela execução em cascata, devido a reativações sucessivas de *Softirqs* por elas mesmas).

No *kernel* padrão não há mecanismos para controlar inversões de prioridades em nível de interrupção. Identificar e quantificar essa interferência é útil quando se quer analizar a adequação do kernel padrão a determinadas aplicações, onde se pretende utilizar o Linux com o mínimo de modificações necessárias (ou idealmente, com nenhuma modificação). Busca-se identificar em que circunstâncias haverá mais ou menos interferência, ou seja, em que situações determinadas fontes causarão mais interferência (em termos de tratadores de interrupção e adiamento de trabalho) e o quanto de interferência será propagada no tempo de execução da aplicação.

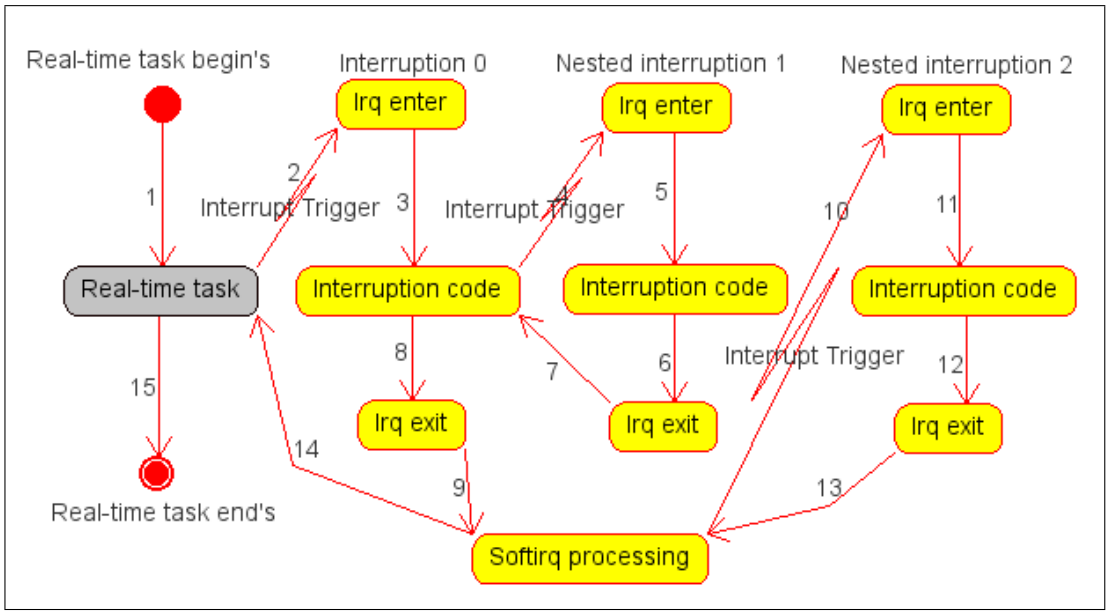

Figura 1. Exemplo de preempção de tarefa por interrupções

A Figura 1 exemplifica a interação entre *Hard irqs*, *Softirqs* e tarefas no Linux. Na figura, a tarefa com prioridade de tempo real (poderia também ser uma tarefa normal) é preemptada pela *Interruption 0*, esta por sua vez, no decorrer de sua execução, é preemptada pela interrupção aninhada *Nested Interruption 1*. Quando a *Nested Interruption 1* finalmente termina sua execução, *a Interruption 0* é retomada e em seu término inicia-se a execução de *Softirqs* pendentes, que logo também sofre preempção pela *Nested Interruption 2* que executará até o fim, e então retornará para o contexto de execução das *Softirqs*. No final desta, será retornado ao contexto em que estava a *Interruption 0*, e então finalemte retornar para a tarefa de tempo real que foi preemptada. Esta situação é apenas um exemplo de prempção de tarefas por mecanismos de tratamento de interrupção (neste, caso com aninhamentos).

4. Monitoração do Kernel

Para os testes foi desenvolvido um patch específico, pois a solução presente no *kernel* (via /proc/stat) para registro de tempo de interrupções e *Softirqs* é baseada em *jiffies*, o que para este estudo não fornece uma granularidade de tempo fina o suficiente. *Jiffies* são incrementados a uma taxa de 100, 250 ou 1000 Hz, ou seja, a precisão dos resultados depende da configuração utilizada. O *Patch* desenvolvido realiza contagens de tempo utilizado uma instrução da arquitetura x86 chamada *rdtsc*, que faz a leitura do *timestamp counter*, um registrador incrementado a cada ciclo de *clock* do processador. A conversão de *clocks* para microsegundos pode ser feita dividindo a quantidade total de *clocks* pela freqüencia do processador em MHZ.

Para quantificar a interferência causada pelas *Hard irqs* e *Softirqs*, foi necessário modificar o *kernel* do Linux (mais especificamente a versão 2.6.28.2 [Torvalds 2008]) para registrar e fornecer dados relevantes sobre a execução das mesmas. As modificações levaram em conta conta, e tiveram como base, o multiprocessamento simétrico (mas podem ser perfeitamente utilizadas em sistemas com apenas um processador). Sem perda de generalidade, foi utilizada a arquitetura x86 como foco de desenvolvimento. Os dados

que o *kernel* passou a fornecer, por processador, são: a quantidade de tempo (em ciclos de *clock*) que o sistema gastou em contexto de interrupção e o tempo total consumido com processamento das *Softirqs*.

A separação do tempo de *Hard irqs* e *Softirqs* e gerada pela seguinte formula: Tempo de *Hard irqs* = Tempo total gasto em contexto de interrupção – Tempo consumido por *Softirqs*. A interface do *patch* com o espaço de usuário foi implementada através de um pseudo *device driver* de caractere carregável via módulo.

5. Descrição das Experiências

O código a seguir foi utilizado na realização das experiências:

```
void do_some_work(){
    int i, j, k, l, max;
    int prod = 0;

    max = 100;
    for(i = 0; i < max; i++){
       for(j = 0; j < max; j++){
          for(k = 0; k < max; k++){
             for(l = 0; l < max; l++){
                prod = i*j*k*l;
             }
          }
       }
    }
}
```

As experiências foram realizadas em um notebook com as seguintes configurações: processador AMD Turion 64 X2 TL-50, 1600MHz. 2 Gb de memória ram, distribuição OpenSUSE 11.1 com o *kernel* 2.6.28.2 (a versão modificada).

5.1. Recursos Utilizados

SMP IRQ Affinity: Desde a versão 2.4 do *kernel*, é possível definir *affinity mask's*, que são máscaras de bits que especificam quais CPUs poderão atender determinadas fontes de interrupção. Não é possível desabilitar todas as CPUs de uma determinada fonte. Se o controlador de interrupções não suportar *IRQ affinity*, as máscaras não mudarão do valor padrão 0xffffffff. As máscaras são alteradas manipulando os arquivos dos subdiretórios do diretório /proc/irq.

SMP CPU Affinity: Outro recurso implementado no Linux é a possibilidade de definir *affinity mask's* também para tarefas. Nos testes, esse recurso foi utilizado para fixar em que processador a tarefa com prioridade de tempo real deveria executar, e posteriormente levar em conta apenas as *softirqs* que foram processadas neste mesmo processador, ou seja, apenas as que realmente causaram interferência. Entretanto, esse não é um recurso amparado pelo padrão POSIX, então deve ser usado como um extensão puramente GNU.

Manipulação da Política de Escalonamento: Segundo o padrão POSIX.4, é possível que uma tarefa determine sua política e prioridade de escalonamento, desde que o processo tenha privilégios para isso. A política utilizada foi a SCHED_FIFO, que é uma política de escalonamento para tempo real.

5.2. Condições das Experiências

Os testes realizados tem por objetivo mostrar a interferência imposta pelas *Softirqs* e *Hard irqs* em uma tarefa de tempo real. A exposição tem por objetivo evidenciar que o problema realmente existe.

Para sobrecarga do sistema optou-se pelo tráfego intenso de rede no dispositivo *ethernet* através de *downloads* (as velocidades estavam entre o limite mínimo de 6 MB/s e máximo 9 MB/s) de arquivos provenientes da internet, pois este se baseia em um subsistema do Linux que faz uso intenso das *Sofirqs*. Aqui nota-se outro ponto que dificultaria a análise quantitativa precisa dos resultados: a quantidade de interferência imposta pelas *Hard irqs* e *Softirqs* sofre influência direta e não previsível da rede e do servidor que está enviado pacotes para máquina de testes. Para cada configuração de teste, foram executados 100 testes.

6. Resultados das Experiências

Cada teste é representado por uma tabela com dados estatísticos. Um gráfico de pizza mostra as proporções de tempo que compõem o tempo total da tarefa.

- Hard irq time: O total de tempo da tarefa ocupado com tratadores de interrupção (e a infraestrutura associada).
- Softirq time: O total de tempo da tarefa ocupado com processamento de *Softirqs* (e a infraestrutura associada).
- Task time: o tempo total de execução da tarefa (tempo de computação referente a um período).
- Task – (Hard + Soft): tempo de execução subtraído dos tempos de *Hard irqs* e *Softirqs* (se aproxima do tempo de computação mínimo possível para uma tarefa completar sua execução).

	Hardirq time (μs)	Softirq time (μs)	Task time (μs)	Task-(Hard+Soft) time (μs)
Média	1017	42	566289	565230
Var.	7102	87	11082	148
D.P.	84	9	105	12
Max	1421	88	566796	565289
Min	978	37	566247	565223

Tabela 1. Estatísticas do teste 1

Teste 1: Segundo a figura 6 (a) e a tabela 1, com carga normal, apenas rodando *daemons* do Linux e o servidor X/KDE, o sistema impôs um *overhead* de tempo (que para uma tarefa de tempo real, é considerado interferência) aproximado de 0.18% devido a *Hard irqs* e 0.01% para *Softirqs*, o que é razoavelmente baixo, se levarmos em conta

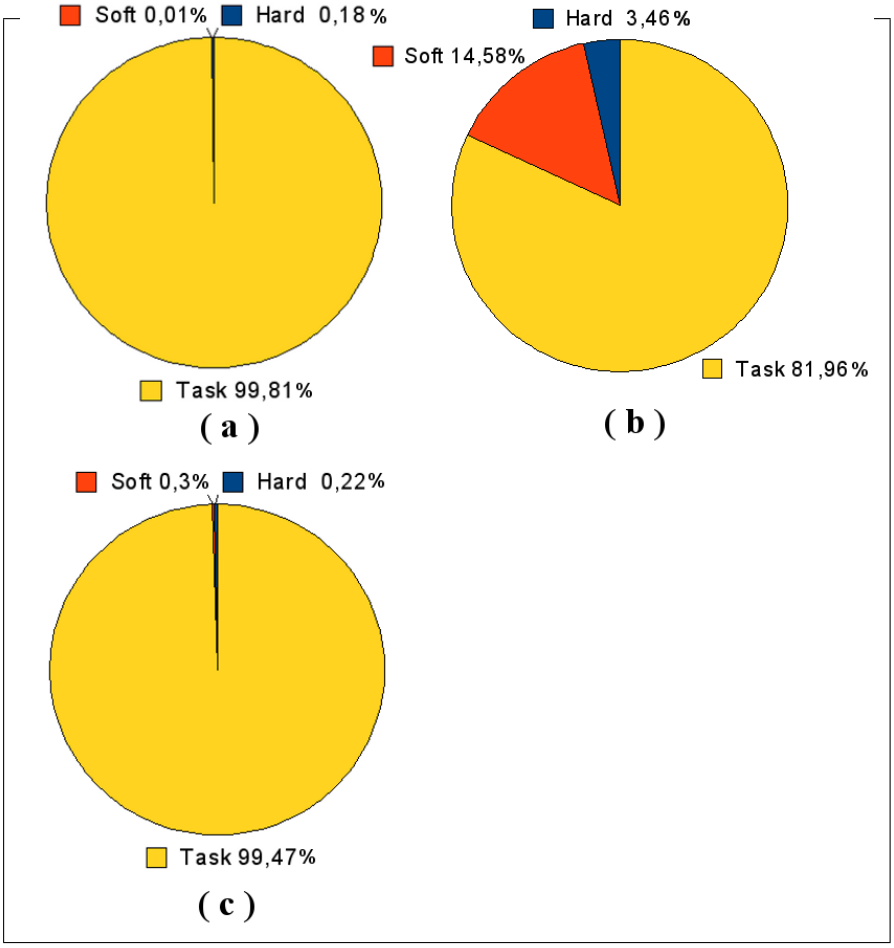

Figura 2. Gráfico das proporções que compõem o tempo de resposta dos experimentos. Em (a), (b) e (c) experiências 1, 2 e 3, respectivamente

o tempo total. Isso nos diz que, em um sistema com baixa taxa de E/S, tarefas com prioridade de tempo real sofrem pouca interferência, ou seja, se houver garantias que o sistema permanecerá neste estado, será viável a execução de tarefas de tempo real.

	Hardirq time (μs)	Softirq time (μs)	Task time (μs)	Task-(Hard+Soft) time (μs)
Média	23919	100779	691219	566520
Var	21957018	444628175	667396553	60671
D.P.	4685	21086	25834	246
Max	30012	132264	725762	567279
Min	2899	8715	577040	565421

Tabela 2. Estatísticas do teste 2

Teste 2: Neste caso, tanto a tarefa quanto o tratador de interrupções da interface de rede, foram fixados à CPU 0. Com relação a figura 6 (b) e a tabela 2, o sistema, sob o tráfego intenso de pacotes na interface de rede, manteve a interferência de *Hard irqs* em uma faixa próxima a 3.46% e *Softirqs* 14.58%. Aqui pode-se observar o quão críticas as *Softirqs* e *Hard irqs* podem ser em determinadas situações. A título de ilustração, a plotagem das medições dos testes está exposta na figura 3. Esta figura nos mostra

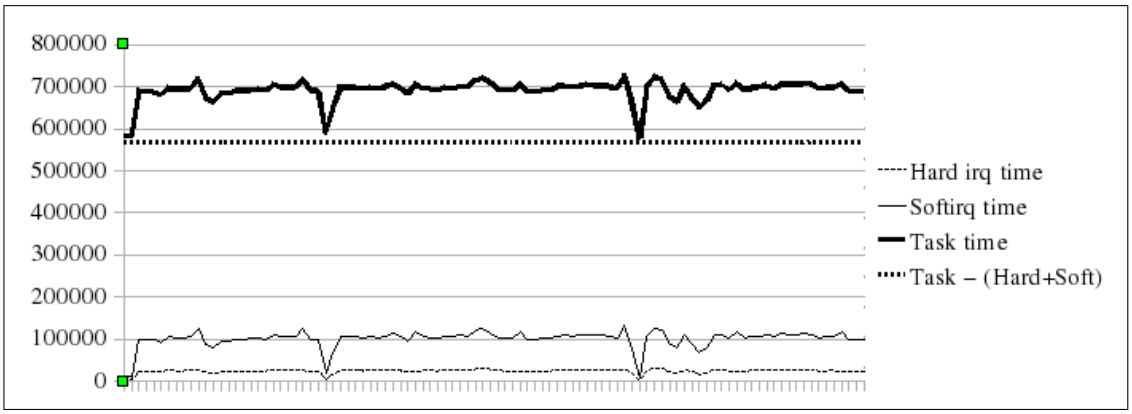

Figura 3. Gráfico da linha de execução teste-a-teste da experiência 2

claramente como as softirqs modelaram a curva de execução da tarefa. Outra informação importante pode ser obtida (ainda na figura 3): A linha Task - (Hard+Soft) mostra o tempo mínimo de resposta da tarefa, em contraste com a linha Task time que mostra o tempo de resposta real, que sofreu influências de *Hard irqs* e *Softirqs*, evidenciando um não determinismo que o sistema impôs à tarefa. Neste teste, não é possível executar uma tarefa deterministicamente como no anterior.

	Hardirq time (μs)	Softirq time (μs)	Task time (μs)	Task-(Hard+Soft) time (μs)
Média	1277	1732	568229	565218
Var	30504	27117587	27458596	188
D.P.	174	5207	5240	13
Max	1905	25236	592115	565291
Min	960	40	566228	565197

Tabela 3. Estatísticas do teste 3

Teste 3: Neste teste, a máscara de afinidade das interrupções de rede não foi alterada, pois o objetivo era verificar como o sistema sozinho iria responder às interrupções tendo uma tarefa de tempo real fixada na CPU 0. Como se pode observar na tabela 3 e na figura 6 (c), mesmo com sobrecarga, o sistema se comportou deterministicamente. Mesmo neste caso, com as interrupções de rede automaticamente sendo direcionadas para a CPU 1, não há garantias de não preempção da tarefa de tempo real por estas, mas esta garantia pode ser imposta com o ajuste da mascara de afinidade do tratador de interrupção da interface de rede, que embora seja uma solução viável, depende do uso de um sistema SMP com um controlador de interrupção que suporte afinidade.

7. PREEMPT-RT

O PREEMPT-RT [Molnar 2005, Rostedt and Hart 2007] é um *patch* para o *kernel* do Linux, mantido por Ingo Molnar, que tem como um dos principais objetivos permitir a execução determinística de tarefas com alta prioridade no *kernel* do Linux, através de um *kernel* com baixas latências e totalmente preemptável (com exceção do código de interrupção – dependente de arquitetura).

Por padrão, o PREEMP-RT mesmo já sendo um projeto maduro, ainda não vem totalmente inserido no *kernel vanilla* do Linux, pois este *patch* introduz modificações muito profundas, e que ainda são gerenciadas com truques de compilação. Isto torna sua aceitação difícil e lenta, mais tudo indica que logo ele será fundido novamente com o *mainline* do Linux, pois já está sendo utilizado por muitos distribuidores de software, o que indica ser uma solução madura e estável.

7.1. Repetição de Experimento Sobre um Kernel com o PREEMPT-RT

Com o objetivo de comparar o Linux *mainline* com o Linux *real-time* em termos de desempenho de tempo real, o *patch* anteriormente desenvolvido foi portado para este último (mais especificamente para a versão 2.6.29-rc7 adicionada do patch PREEMPT-RT patch-2.6.29-rc7-rt1). O teste foi repetido na mesma circunstância de carga dos teste 2.

	Hardirq time (μs)	Softirq time (μs)	Task time (μs)	Task-(Hard+Soft) time (μs)
Média	1217	25	566467	565224
Var	663	14	790	8
D.P.	25	3	28	2
Max	1318	36	566591	565242
Min	1169	18	566419	565220

Tabela 4. Estatísticas do teste 4

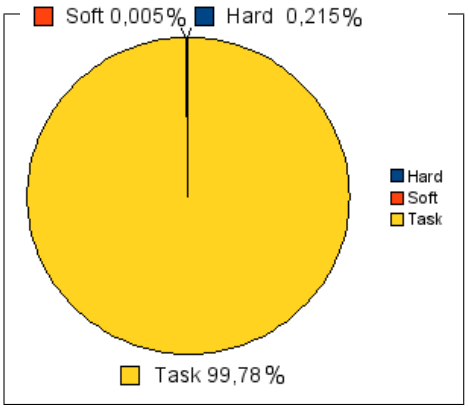

Figura 4. Gráfico das proporções que compõem o tempo total do teste 4

Teste 4: Neste teste (figura 4 e tabela 4), o sistema foi submetido à mesma sobrecarga utilizada no teste 2 e mesmo estando tanto a tarefa de tempo real, quando a interrupção fixadas à CPU 0, o sistema apresentou um considerável grau de determinismo (que pode ser observado pela linearidade da figura 5), com os tempos de resposta de cada teste aparecendo de forma constante no gráfico. Isto indica que o sistema se comportou como se não estivesse sendo submetido à sobrecarga. Este efeito de determinismo na interferência ocorre pois os *threaded interrupt handlers*, e as *Softirqs* estão executando também sob a política de tempo real, mas com prioridade inferior a da tarefa de teste (metade da prioridade máxima para as *Softirqs*). As máscaras fixadas para a CPU 0 também simulam o comportamento do sistema caso só houvesse uma CPU (pois a maioria dos sistemas embarcados ainda não contam com algum tipo de multiprocessamento), mostrando a viabilidade do *patch* para este tipo de sistema também.

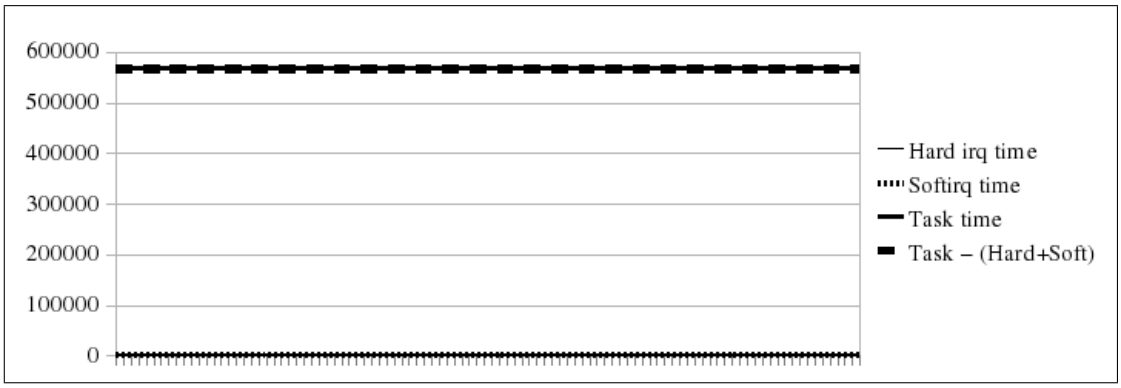

Figura 5. Gráfico da linha de execução teste-a-teste da experiência 4

8. Conclusões

O Linux *mainline*, sendo um sistema de propósito geral, tenta minimizar o tempo médio de resposta do conjunto de tarefas o qual executa. Para isso se faz valer de técnicas de adiamento de trabalho para auxiliar o tratamento de interrupções, tentando assim permanecer com as interrupções desabilitadas o mínimo de tempo possível. Todavia, estes recursos, mesmo sendo benéficos para aplicações normais (neste caso, sem restrições temporais), costumam acarretar certa quantidade de interferência não previsível em tarefas com prioridade de tempo real, mesmo possuindo importância menor que estas.

Segundo os testes executados, percebeu-se que em certas circunstâncias as *Hard irqs* e as *Softirqs* causaram uma quantidade excessiva de interferência em processos de tempo real com prioridade máxima no Linux, levando assim, a um não determinismo no tempo de resposta. Para uma tarefa tempo real crítica (*hard real-time*) poderia levar a uma perda de *deadline*, ou degradação na qualidade de serviço no caso de uma tarefa não crítica (*soft real-time*). Entretanto, caso o sistema desfrute de Multiprocessamento Simétrico (SMP), é possível definir máscaras de afinidade tanto para processos quanto para interrupções (desde que este recurso seja suportado pela arquitetura em uso). Desta forma, é possível fazer um balanceamento manual da carga de interrupções de modo que estas não atinjam diretamente o processador no qual se está executando a tarefa de tempo real, aliviando significativamente as interferências sofridas.

Esta abordagem de balanceamento estático de trabalho entre as CPUs tem inúmeras vantagens, como por exemplo, poder executar o atendimento rápido das interrupções (ou seja, manter alta a reatividade do sistema perante eventos externos) sem que os mecanismos de adiamento de trabalho associados interfiram nas tarefas de tempo real. Estes estão logicamente separados por máscaras de afinidade, ou seja, cada um poderá estar estaticamente habilitado à executar em um subconjunto fixo de CPUs, não competindo diretamente pelo mesmo recurso. Outra vantagem é poder executar aplicações em tempo real que não foram desenvolvidas para este fim (como por exemplo processamento multimídia) apenas ajustando as prioridades, a máscara de *SMP Affinity* e a política de escalonamento, ou seja, sem nenhuma necessidade de recompilação de código. Todavia, esta abordagem necessita de uma rigorosa avaliação do sistema e do ambiente de modo a identificar e isolar todas as possíveis interrupções nocivas a tarefa de tempo real. O que muitas vezes pode levar a um certo desperdício de tempo de processamento, pois haverá a possibilidade de se estar negando a execução de um tratador de interrupção a uma CPU

com baixa carga de processamento, apenas por ter uma tarefa de tempo real periódica afixada nesta.

Contudo, como alternativa ao Linux *mainline*, foi apresentado o PREEMPT-RT, que é uma solução de tempo real para Linux que tem o diferencial de não ser tão intrusiva quanto a maioria das outras solução (como o RTAI e o XENOMAI por exemplo). Pode-se ter o mesmo ambiente operacional (com TCP/IP, acesso nativo a dispositivos, interface gráfica e outros) do Linux convencional, mas com toda uma infraestrutura voltada para um ambiente altamente preemptivo, com herança de prioridade nos *spinlocks* e *mutexes*, e reduzidos trechos de código em tarefas não gerenciadas pelo escalonador. De acordo com os testes, o Linux com PREEMPT-RT conseguiu manter o tempo de resposta da tarefa constante. Isto independentemente de se estar ou não impondo algum tipo de carga de interrupção, mesmo com a tarefa de tempo real e a interrupção fixadas à mesma CPU, o que no Linux padrão provocou flutuações significativas no tempo de resposta.

De acordo com o exposto acima, pode-se concluir que o Linux padrão oferece uma alternativa competitiva como solução de tempo real em sistemas com múltiplos processadores, através da sua interface POSIX, e de extensões não-POSIX que tentam preencher a não abrangência do padrão no que diz respeito a Multiprocessamento Simétrico. Mas, se o interesse recair em um sistema de tempo real completo, o PREEMPT-RT fornece uma opção madura e estável, pois não depende de ajustes de máscaras de afinidade para prover o determinismo no tempo de computação que tarefas de alta prioridade necessitam.

Referências

Dozio, L. and Mantegazza, P. (2003). "Linux Real Time Application Interface (RTAI) in low cost high performance motion control". *Proceedings of the conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*.

Gerum, P. (2004). "Xenomai - Implementing a RTOS emulation framework on GNU/Linux".

Love, R. (2005). *"Linux Kernel Development"*. SAMS, second edition.

Molnar, I. (2005). "PREEMPT-RT". http://www.kernel.org/pub/linux/kernel/projects/rt - Last access 01/21, 2009.

Molnar, I. (2007). "Modular Scheduler Core and Completely Fair Scheduler [CFS]". http://kerneltrap.org/node/8059 - Last access 03/26, 2009.

POSIX.13 (1998). *"IEEE Std. 1003.13-1998. Information Technology -Standardized Application Environment Profile-POSIX Realtime Application Support (AEP)"*.

Regnier, P., Lima, G., and Barreto, L. (2008). "Evaluation of interrupt handling timeliness in real-time Linux operating systems". *SIGOPS Oper. Syst. Rev.*, 42(6):52–63.

Rostedt, S. and Hart, D. V. (2007). "Internals of the RT Patch". *Proceedings of the Linux Symposium,*, pages 161–172.

Torvalds, L. (2008). "Linux Kernel Version 2.6.28.2". http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.28.2.tar.bz2 - Last access 03/21, 2009.