Coping With Interrupt Execution Time in RT Kernels: a Non-Intrusive Approach

Luca Abeni
ReTiS Lab, Scuola Superiore S. Anna, Pisa
luca@sssup.it

Abstract

This paper describes an on-going work focused on using Adaptive Reservations to cope with unpredictabilities due to the interrupt processing times in real-time kernels. Using this approach, the execution time stolen by interrupt processing is modeled as a variation in the application execution times, and an adaptive scheduling mechanism automatically compensates its effects. The proposed approach has been implemented in Linux/RK, and some preliminary experimental results are presented.

1 Introduction

In the last years, there has been a growth of interest in running applications with real-time requirements (such as streaming applications, software mixers, and so on) on top of desktop Operating Systems (OSs). For this reason, a lot of effort has been spent in providing general purpose desktop OSs (such as Linux, FreeBSD, or Windows) with real-time capabilities.

However, omitting some notable exceptions, these works mainly focused in implementing real-time scheduling algorithms for the CPU and other system resources inside the OS kernel. Most of these algorithms are based on some very stringent assumptions, such as the full preemptability of each resource (that is, the fact that each resource can be preempted at any instant in time), that may be violated by the most diffuse OS structures. For example, using the monolithic kernel structure upcoming hardware events like interrupts can "steal" some CPU time to the real-time tasks, compromising the whole real-time guarantee mechanism that the kernel may implement.

In other words, the theoretical guarantee that can be performed based on well established results from real-time theory does not match with the practical results, due to the kernel overhead caused by nonpreemptable sections or interrupt processing. Of course, these effects can be accounted in the guarantee [6], or can be mitigated [7, 4], performing for example a "soft interrupt scheduling" [5, 3]. However, the first solution requires a precise knowledge of the ISR execution times and of the interrupt arrival times, and this information is not available on a general purpose workstation. On the other hand, the second solution is "too intrusive", meaning that it requires heavy modifications to the kernel and to the drivers.

In [9], the authors proposed a completely different solution, based on adapting the scheduling parameters in order to respect the real-time constraints even when some kernel overhead is present. That solution appears to be very effective, and it results more usable than the previously presented ones. However, it still requires to instrument the kernel code (in Windows, the ISR and the DPC code), hence the kernel needs to be modified. In this paper, a similar solution, based on Adaptive Reservations [1] is presented. Adaptive reservations can be implemented in user space over every kernel that provides temporal protection in CPU scheduling, without requiring any additional modification to the kernel.

2 The Problem

As stated in the introduction, one of the biggest problems in the real-time extensions to workstation operating systems is that they generally consider the CPU as the only hardware resource in the system (hence, as the only resource that has to be scheduled), or they generally consider hardware resources as independent. However, a modern PC is connected to a lot of peripherals, which can be considered as hardware resources that the OS kernel has to manage. In most cases, those resources can produce events (in the form of hardware interrupts), and the kernel has to properly manage them, or forward them to an appropriate handler tasks.

In a μkernel based system, a hardware interrupt can be converted in an IPC to a server task, that will properly handle the hardware device; in a multithreaded kernel, a kernel thread can be used to properly serve the interrupt, whereas in a monolithic kernel (such as Linux or *BSD) the handler will run immediately before process dispatching.

Independently from the kernel structure, a hardware interrupt is generally served in two phases:

- a short **Interrupt Service Routine (ISR)** is responsible for acknowledging the hardware interrupt mechanism and activating a proper handling mechanism;
- a longer routine, activated by the ISR, is responsible for correctly handling the data to be received or sent through the device.

If a μkernel based structure or a multithreaded kernel is used, the longer routine handling the device data runs in a server task or in a kernel thread that can be scheduled like all the other tasks in the system. This solution can present a slightly higher overhead, and requires a more careful synchronization, but permits to correctly account the handler code in a time sensitive system. In facts, the handler code requires some CPU time to execute, and it must be correctly accounted in order not to break the system's guarantees.

In order to better understand the problems deriving from an incorrect accounting, let's consider the traditional monolithic kernel structure, upon which Linux, *BSD, and many other kernels are based. A single program, the *kernel*, running in privileged mode, abstracts the hardware providing a high level Application Binary Interface (ABI). Applications cannot directly access hardware resources, since protection is enforced, but must require such an access to the kernel.

The kernel is implemented as a single-threaded program, hence only one single execution flow can be active inside the kernel at each time. The kernel responds to two different kinds of requests, coming from up (application requests in form of system calls - using the INT instruction in the x86 CPU family) or down (hardware interrupts).

Application requests are often called *top halves* in Unix terminology. As explained above, hardware requests are served in two phases: when an interrupt fires, an ISR starts and is responsible for acknowledging the hardware device. After that, it activates a *bottom half* (in Linux terminology), or *soft interrupt* (using the *BSD terminology), that is responsible for finishing to serve the hardware request, and will be executed immediately before switching to user mode again.

Most of the handler code runs in bottom halves, that are executed by the dispatcher before the scheduled process, and are not preemptable with respect to the application processes. This fact can introduce three sources of unpredictability:

1. the handler code is executed at apparently random times (depending on the interrupts' arrival pattern) and *is not scheduled* (it always has priority over application processes), introducing anomalies in the CPU scheduling;

2. as a result, the handler execution time is accounted to the interrupted process, and is seen by user application as *stolen time*;

3. bottom halves are not preemptable, violating one of the assumptions of real-time scheduling theory.

```
init = rdtsc();
for (i = 0; i < MAX; i++) {
    for (j = 0; j < COUNT; j++) {
        for (k = 0; k < 100; k++) { /* Just to spend some time...  */
            time = rdtsc();
        }
    }
    timevect[i] = CLOCK2USEC(time - init);
    task_endcycle(); /* Blocks until the next period */
}
```

Figure 1: The test process

As a result of these scheduling anomalies, the real-time guarantees provided by the system may be broken. From a practical point of view, the system behaves like if some execution time (the bottom half execution time) has been stolen to application processes, hence this problem will be referred as the *stolen time* problem.

In order to show the impact of this problem, some experiments have been performed using Linux/RK, a Resource Kernel based on Linux. A Resource Kernel in general permits to reserve an hardware resource to a process: based on some *reserve* abstractions, a process can be guaranteed to receive the resource for a time Q each period T. The version of Linux/RK used for these experiments provides predictable guarantees for CPU reservations, outgoing network reservations, and disk reservations, but does not properly account the time stolen by the bottom halves.

The influence of the bottom halves on the CPU scheduling can be easily seen by simply causing a lot of bottom halves execution and measuring the impact on the execution of a reserved process, as shown by the following experiments. First of all, the periodic process shown in Figure 1 has been run with period $T = 20ms$ on an AMD-K6 at 333 MHz, attached to a proper CPU reservation. Since the `COUNT` constant is tuned so that the `j` loop takes about $4ms$, when attached to a $(4ms, 20ms)$ reservation this process does not miss any deadline. In fact, the difference between two consecutive values of `timevect` (referred as *job inter finishing time* in this paper) is about constant, and equal to $20ms$ (the process & reserve period), as shown in Figure 2.

After that, a heavy network traffic has been sent to the test machine, in order to increase the CPU time consumed by bottom halves. When the network load is increased, the bottom

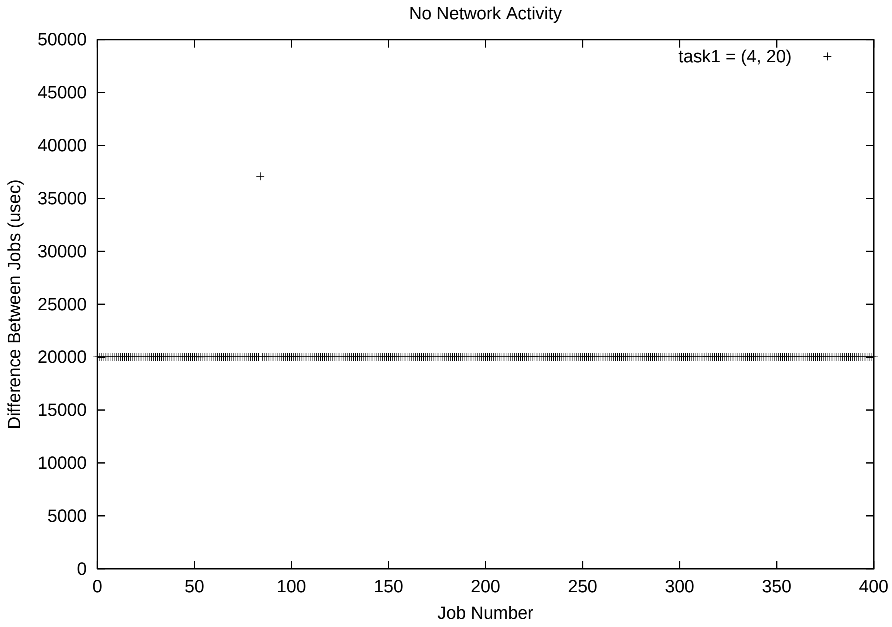

Figure 2: Reserved process running in regular network load conditions: job inter finishing times.

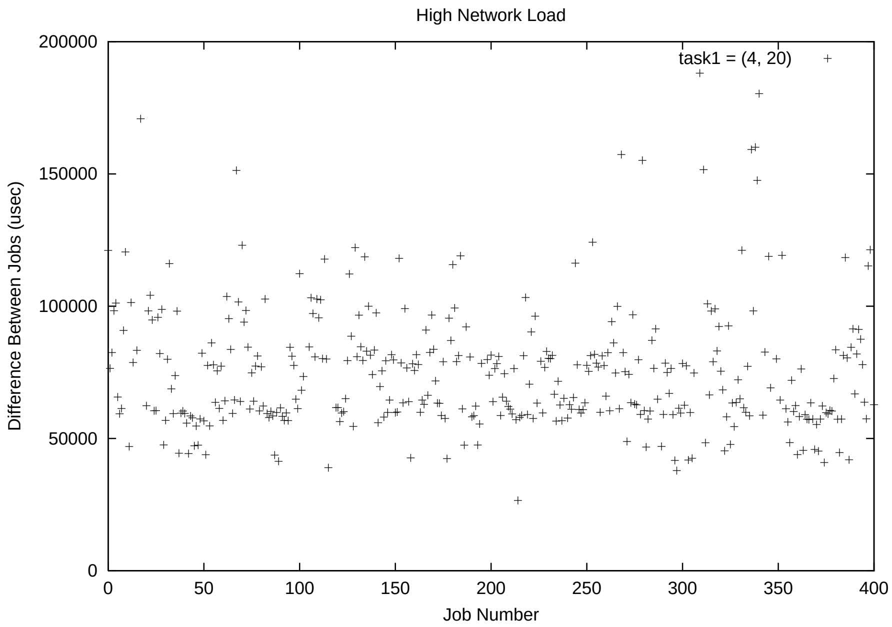

Figure 3: Reserved process running in high network load conditions: job inter finishing times — the process misses deadlines.

halves steal execution time to the reserved process, hence the difference between two consecutive values of the `timevect` array increases, and the process starts to miss deadlines, as shown in Figure 3.

3 A Possible Way Out

If a reservation based scheduler is used, a possible solution to the "stolen time problem" could be to use the *augmented reservation* abstraction [9], that permits to resize the system reservations in order to compensate the effects of the stolen time.

The augmented reservations approach results to be very effective and easy to implement, but it requires to monitor the time stolen by DPCs or bottom halves (hence, it requires additional modifications to the OS kernel), in order to sum it to the reserved time. This requirement is due to the fact that augmented reservations have been designed to support a generic task model; if, on the other hand, the real-time task model is used, the requirement of instrumenting the kernel can be relaxed. In fact, using the real-time task model each task is divided in jobs, and each job is characterized by an absolute deadline that can be used to monitor application performance. In this way, an implicit monitoring of the effects of the interrupt handlers execution can performed by simply measuring the number of missed deadlines, and the DPC or bottom half time does not have to be explicitly monitored.

This idea is used by *Adaptive Reservations*, presented in [1]: the adaptive reservation abstraction was originally developed in order to cope with tasks characterized by unknown or highly variable execution times, but it can be successfully used to mitigate the effects of stolen time. In fact, the time stolen by ISRs and bottom halves can be modelled as a variance in tasks' execution times, and adaptive reservations will properly cope with it.

Due to space constraints, a complete description of the adaptive reservation abstraction and of its implementation is outside the scope of this WiP presentation (that is more focused in showing how adaptive reservations cope with stolen time): interested readers are invited to consult the cited paper. In order to understand the use of adaptive reservations in this work, only few basic concepts are recalled:

- A process attached to an adaptive reservation is initially reserved Q time units each period T, where T is the process period and Q is initialized to an arbitrary value;
- if the process misses some deadline, the reserved time Q is increased;

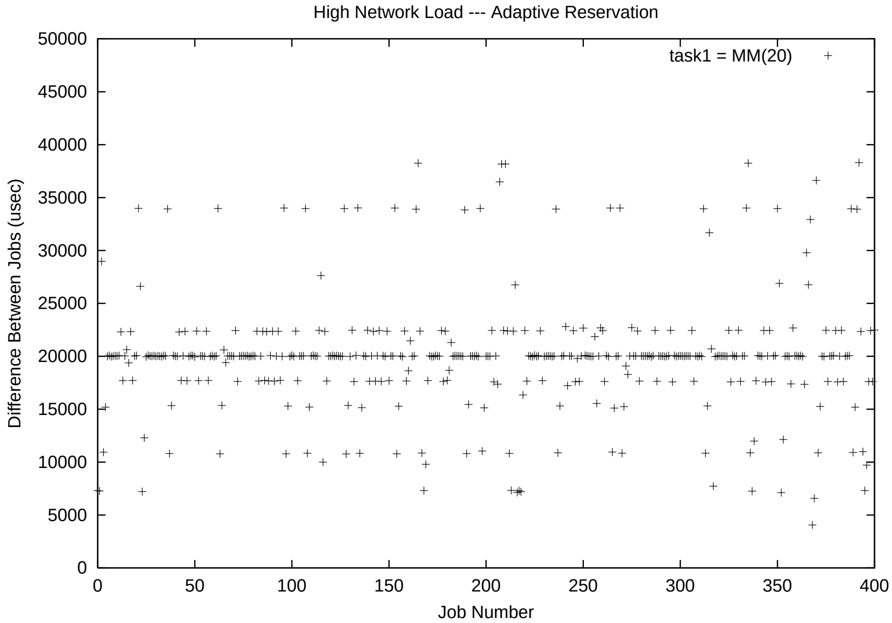

Figure 4: Adaptive Reservations in high network load conditions: job inter finishing times — the number of missed deadlines is controlled.

- if the process does not miss deadlines for a sufficiently long time, its reserved time Q is decreased.

When the network load increases, the bottom halves begin to consume a significant amount of CPU time, stealing it to reserved processes. Hence, a reserved process will miss some deadline, and if the process is attached to an adaptive reservation its reserved time will be increased. In this way, the amount of time reserved to a process increases when the network traffic increases, compensating the effects of the bottom halves execution.

Note that, in contrast with augmented CPU reservations, adaptive reservations can be implemented in user space, without requiring modifications to the kernel. The only requirement is that the kernel provides temporal protection in the CPU scheduler; as a proof of concept, adaptive reservations have been implemented through a portable QoS Manager, that has been ported on the HARTIK kernel [2] and Linux/RK [8]; in this work, the RK version has been used.

In order to test how adaptive reservations can solve the problem highlighted in Figure 3, the experiment described in Section 2 has been repeated attaching an adaptive reservation with period $20ms$ to the user process. As a result, the process parameters were adapted so that the number of missed deadlines resulted to be controlled to 0 after a short transient.

In fact, the difference between two consecutive values of `timevect` resulted to be controlled below $40ms$, as shown in Figure 4.

4 Conclusion and Future Work

In this paper, the stolen time problem has been discussed, and a non intrusive solution to it has been presented. The proposed solution, based on adaptive reservations, is claimed to be non intrusive because if the OS kernel already provides temporal protection then adaptive reservation can be fully implemented in user space.

The first results seem to be very promising, but this is still a work in progress, and more experiments have to be performed as a future work in order to verify the impact of the interaction between network flows and reserved applications.

References

[1] Luca Abeni and Giorgio Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.

[2] Luca Abeni and Giorgio Buttazzo. Support for dynamic QoS in the HARTIK kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

[3] Peter Druschel and Gaurav Banga. Lazy reciver processing (LRP): A network subsystem architecture for server systems. In *Proceding of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, Oct 1996.

[4] A. Indiresan, A. Mehra, and K. G. Shin. Receive livelock elimination via dynamic interrupt rate control. Technical report, University of Michigan, June 1997.

[5] Kevin Jeffay, F.D. SMith, A. Moorthy, and J.H. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[6] Kevin Jeffay and Donald L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *IEEE Real Time System Symposium*, pages 212–221, 1993.

[7] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3), August 1997.

[8] Ragunathan (Raj) Rajkumar, Luca Abeni, Dionisio de Niz, Sourav Ghosh, Akihiko Miyoshi, and Saowanee Saewong. Recent developments with linux/rk. In *Proceedings of the Second Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[9] John Regehr and John A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.