A Measurement-Based Analysis of the Real-Time Performance of Linux [*]

Luca Abeni[†], Ashvin Goel, Charles Krasic, Jim Snow, Jonathan Walpole
luca@sssup.it, {ashvin, krasic, jsnow, walpole}@cse.ogi.edu
Department of Computer Science and Engineering
Oregon Graduate Institute, Portland

Abstract

This paper presents an experimental study of the latency behavior of the Linux OS. We identify major sources of latency in the kernel with the goal of providing real-time performance in a widely used general-purpose operating system. We quantify each source of latency with a series of micro-benchmarks and also evaluate the effects of latency on a time-sensitive application. Our analysis shows that there are two main causes of latency in the OS: timer resolution and non-preemptable sections. Our experiments show that in the standard Linux kernel the timer resolution latency is predominant, and generally hides the non-preemptable section latency. We use accurate timers to reduce timer resolution latency and then analyze the non-preemptable section latency for several variants of Linux.

1. Introduction

In the last several years, there has been an explosive growth in interest in supporting multimedia applications such as video streaming programs, software audio mixers, etc., on general-purpose operating systems. These multimedia applications, and soft real-time applications in general, are characterized by *implicit temporal constraints* that must be satisfied to provide the desired QoS. We thus call these applications *time-sensitive* applications.

To support time-sensitive applications, a general-purpose kernel must respect the application's temporal constraints and hence a predictable schedule is needed. Unfortunately, general-purpose kernels often generate a schedule that is different from the expected one due to various reasons such as the implementation specifics of the kernel. This paper evaluates, measures, and characterizes the temporal behavior of a widely used general-purpose kernel through an extensive set of experiments with the goal of supporting real-time applications on such operating systems (OSs).

In particular, we define a metric called *OS latency* that quantifies the difference between the actual schedule produced by the kernel and the ideal schedule. Based on this definition, we perform a comprehensive, quantitative evaluation of latency in Linux [21]. We chose Linux because it is widely used, supports most commonly available hardware and is distributed under an open source license [6] which enables researchers to easily experiment with it.

We identify several sources of the OS latency, the two most important sources being *timer resolution* and *non-preemptive sections* in the kernel or in the drivers. We designed a set of micro-benchmarks and have used these benchmarks to systematically quantify each source of latency in Linux. In addition, we compare the latency behavior of the standard Linux kernel with the behavior of some modified versions of the kernel. We show that the application of some well known real-time concepts such as full kernel preemptability can greatly improve the real-time performance of Linux.

We also quantify the effects of the OS latency at the application-level by instrumenting a well-known audio/video application and thus experimentally evaluate how well a general-purpose Linux kernel can support the real-time performance needs of multimedia applications.

The main contribution of this paper is a characterization of the temporal behavior of a general-purpose kernel. We believe that this study is important because it enables using real-time analysis for such systems. The results contained in this paper complement many of the results obtained in real-time research, in that they help focus attention on the major sources of latency in practice, and hence help us move towards the goal of realizing real-time behavior in a widely used operating system such as Linux.

The rest of the paper is organized as follows. Section 2 formally defines the OS latency and investigates the factors

[*] This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, NSF Grant CCR-9988440 and EIA-0130344, and by Intel.

[†] Luca Abeni is affiliated with ReTiS Lab, Scuola Superiore S. Anna, Pisa, Italy.

that contribute to it. In Section 3, we describe the experimental setup for evaluating the various components of the OS latency. Section 4 presents the experimental results, and in Section 5 we show how OS latency affects a media application running on Linux. Finally, in Section 6 we present related work and in Section 7 we state our conclusions.

2. The OS Latency

The main objective of Linux in terms of performance is to provide fairness and high throughput, i.e., minimizing the average execution time experienced by concurrently executing processes. Until recently, Linux has not focused on time-sensitive applications, which are characterized by temporal constraints. Such applications may require periodic execution where, for example, the period is derived from the frame rate of an audio/video stream, or they may require response in a short time to external events such as the arrival of a network packet.

In this paper, we use *OS latency* as a metric to evaluate the OS support for time-sensitive applications. We define the OS latency as follows:

Definition 1 *Let τ be a task*[1] *belonging to a time-sensitive application that requires execution at time t, and let t' be the time at which τ is actually scheduled; we define the OS latency experienced by τ as $L = t' - t$.*

Examples of tasks that need to execute at time t are, for instance, periodic tasks (the task wakes up at time t in response to a periodic event), or tasks that must react in a short time to external interrupts.

2.1. Causes of the OS Latency

OS latency can be caused by several factors. We have identified three major causes of this latency: *timer resolution*, *scheduling jitter*, and *non-preemptable sections*.

Timer resolution latency occurs because kernel timers are generally implemented using a periodic tick interrupt. For example, consider a periodic task τ that needs to execute every $T \mu s$. Typically, the task will be woken up by a kernel timer that is triggered by the periodic tick interrupt with say, period T^{tick}. Hence, a task that sleeps for an arbitrary amount of time T can experience some *timer resolution* latency L^{timer} if its expected activation time is not on a tick boundary.

Scheduling jitter is caused because τ may not be scheduled immediately even if accurate timers ensure that τ enters the ready queue at the correct time. The scheduling jitter experienced by a task τ can be easily eliminated by assigning the highest real-time priority to it.[2] Since real-time scheduling algorithms that reduce scheduling jitter have been widely studied in the literature we will not address this problem in this paper. For the purpose of our experiments we will simply use the highest real-time priority to eliminate the latency caused by scheduling jitter.

A third source of latency, that we call *OS non-preemptable section latency* is caused by non-preemptable sections in the kernel or in the drivers. This component of latency includes Interrupt Service Routines (ISRs) and other kernel constructs such as bottom halves and tasklets. Consider an example where interrupts are disabled at time t. Task τ can only enter the ready queue later when interrupts are re-enabled. In addition, even if τ enters the ready queue at the correct time t and has the highest real-time priority in the system, it may still not be scheduled if preemption is disabled for some reason. In this case, τ will be scheduled when preemption is re-enabled at time t', contributing an OS non-preemptable section latency $L^{np} = t' - t$. This OS non-preemptable section latency includes kernel non-preemptable sections, but also other sources of non-preemptability, which for example may be caused by device drivers, such as ISRs, bottom halves, and so on.

2.2. Analysis of the Latencies

In our experiments, τ is scheduled using the highest real-time priority to eliminate the latency caused by scheduling jitter. Thus, the maximum latency L that τ can experience is equal to the sum of the maximum latencies due to timer resolution and non-preemptable sections ($\max\{L^{timer}\} + \max\{L^{np}\}$). We analyze these two terms separately.

2.2.1 Timer Resolution

Standard Linux timers are triggered by a periodic tick interrupt, which on x86 machines is generated by the Programmable Interval Timer (PIT) and has a period $T^{tick} = 10ms$. As a result, the maximum latency due to the timer resolution $\max\{L^{timer}\}$ is $T^{tick} = 10ms$. Thus, this value can be reduced by reducing T^{tick}. However, decreasing T^{tick} increases system overhead because more tick interrupts are generated. In addition, there is a lower bound on L^{timer} which is equal to the execution time required for servicing the tick interrupt.

The fact that a periodic timer interrupt is not an appropriate solution for a real-time kernel is well known in the literature, and thus most of the existing real-time kernels provide *high resolution timers* based on an aperiodic interrupt source[17]. In an x86 architecture, the PIT or the

[1]In this paper, we use the word "task" to denote either a thread or a process.

[2]Note that using Linux real-time priorities, it is very easy to implement a rate-monotonic policy.

CPU APIC (Advanced Programmable Interrupt Controller present in many modern x86 CPUs) can be programmed to generate aperiodic interrupts for this purpose. We expect that high resolution timers will reduce L^{timer} to the interrupt service time without significantly increasing the kernel overhead, because these interrupts are generated only when a timer expires. In this paper, we consider the timer resolution latency in two different kernels: 1) the standard Linux kernel, 2) a high-resolution timer Linux kernel that we have implemented at OGI. Our experiments in Section 4 show that the resolution of our high-resolution timers lies between $4\mu s$ to $6\mu s$.

2.2.2 OS Non-Preemptable Section Latency

The second term contributing to the maximum OS latency is the OS non-preemptable section latency $\max\{L^{np}\}$. This value depends on the device drivers, but also on the strategy that the kernel uses to guarantee the consistency of its internal structures, and on the internal organization of the kernel. In this paper, we consider latencies of four different variants of the kernel. These kernels use different strategies for protecting their internal structures. These kernels are 1) the standard *Linux* kernel, 2) the *Low-Latency Linux* kernel, 3) the *Preemptable Linux* kernel, and 4) the *Preemptable Lock-Breaking Linux* kernel.

Standard Linux: The standard kernel is based on the classical *monolithic* structure, in which the consistency of kernel structures is enforced by allowing at most one execution flow in the kernel at any given time. This is achieved by disabling preemption when an execution flow enters the kernel, i.e., when an interrupt fires or when a system call is invoked. In a standard Linux kernel, $\max\{L^{np}\}$ is equal to the maximum length of a system call plus the processing time of all the interrupts that fire before returning to user mode. Unfortunately, this value can be as large as $28ms$ as shown in Section 4.

Low-Latency Linux: This approach "corrects" the monolithic structure by inserting explicit preemption points (also called rescheduling points) inside the kernel. In this approach, when a thread is executing inside the kernel it can explicitly decide to yield the CPU to some other thread. In this way, the size of non-preemptable sections is reduced, thus decreasing L^{np}. In a low-latency kernel, the consistency of kernel data is enforced by using cooperative scheduling (instead of non-preemptive scheduling) when the execution flow enters the kernel. This approach is used by some real-time versions of Linux, such as RED Linux [26], and by Andrew Morton's low-latency patch [14]. In a low-latency kernel, $\max\{L^{np}\}$ decreases to the maximum time between two rescheduling points.

Preemptable Linux: The preemptable approach, used in most real-time systems, removes the constraint of a single execution flow inside the kernel. Thus it is not necessary to disable preemption when an execution flow enters the kernel. To support full kernel preemptability, kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptable kernel patch [11] uses this approach and makes the kernel fully preemptable. Kernel preemption is disabled only when a spinlock is held.[3] In a preemptable kernel, $\max\{L^{np}\}$ is determined by the maximum amount of time for which a spinlock is held inside the kernel (maximum size of a kernel non-preemptable section), plus the maximum time taken by ISRs, bottom halves and tasklets.

Preemptable Lock-Breaking Linux: The kernel latency can be high in Preemptable Linux when some spinlock is held for a long time. Lock breaking addresses this problem by "breaking" long spinlocks, i.e., by releasing spinlocks at strategic points. Breaking spinlocks into smaller non-preemptable sections is similar to the approach used by Low-Latency Linux. This approach reduces the size of kernel non-preemptable sections, but, of course, does not decrease the amount of time "stolen" by device drivers.

As a final note, we would like to point out that the preemption patch has been recently accepted in the development (unstable) branch of the Linux kernel, and is now present in version 2.5.4 of the kernel.

3. Experimental Setup

The goal of this paper is to evaluate Linux latency. One method for experimentally measuring the latency is to use a task that invokes `usleep` to sleep for a specified amount of time and then measures the time that it actually slept. The latency L, as defined in Section 2, is then the difference between these two times. Unfortunately, this approach measures the sum of all the latency components and thus does not give us an insight into the causes of latency.

We investigate the individual latency components by measuring each of them in *isolation*, i.e., measure each source of latency while eliminating the others. First, the scheduling jitter is easily eliminated by running the test program at the highest real-time priority. Next, we need to measure timer resolution latency L^{timer} and OS non-preemptable section latency L^{np} in isolation. To measure L^{timer}, we eliminate L^{np} by running the experiment on an

[3]There is also a different patch, from Timesys [9], based on mutexes and priority inheritance instead of on spinlocks.

idle system. To measure L^{np}, we eliminate L^{timer} by using high resolution timers. The following sections describe this approach in more detail.

3.1. Measuring Timer Resolution Latency

The OS non-preemptable section latency L^{np} can be reduced significantly by running experiments on a lightly-loaded system. In this case, few system calls will be invoked and a limited number of interrupts will fire and thus long non-preemptable execution paths or drivers' activations are not likely to be triggered.

The latency L^{timer} can be measured by using a typical periodic time-sensitive application. We implemented this application by running a process that sets up a periodic signal (using the `itimer()` system call) with a period T ranging from $100\mu s$ to $100ms$. The process measures the time when it is woken up by the signal and then immediately returns to sleep. We measured the difference between two successive process activations, which we call the *inter-activation time*. Note that in theory the inter-activation times should be equal to the period T. Hence, the deviation of the inter-activation times from T is a measure of L^{timer}. Since Linux ensures that a timer will never fire before the correct time, we expect this value to be $10ms$ is a standard Linux kernel, and to be close to the interrupt processing time with high resolution timers.

3.2. Measuring OS Non-Preemptable Section Latency

Once the timer resolution latency is eliminated with high resolution timers, we can measure L^{np} in isolation. Unfortunately, a periodic process is not suitable for measuring this latency. For example, to measure the effects of disabling preemption for a time S, the latency must be sampled with a period $T \ll S$ or else the non-preemptive code could execute between two consecutive measurements. More precisely, if $\mathcal{L}$ is the measured latency, then $\mathcal{L} \leq L^{np} \leq \mathcal{L}+T$. Hence, to reliably measure L^{np}, the test task should have a period T such that $T << L^{np}$. In practice, this requirement is hard to achieve and thus we use an aperiodic test application that uses the `usleep()` call.

The test task is based on a loop that:

1. reads the current time t_1
2. sleeps for a time T
3. reads the time t_2, and computes $L^{np} = t_2 - (t_1 + T)$

Times t_1 and t_2 are read using the Pentium Time Stamp Counter (TSC), a CPU register that is increased at every CPU clock cycle and can be accessed in a few cycles. Hence, the measurements introduce very low overhead and are very accurate.

We investigated how various system activities contribute to L^{np} by running various background tasks. The following tasks are known to invoke long system calls or cause frequent interrupts and thus they trigger long non-preemptable sections either in the kernel or in the drivers (as explained in Section 2).

Memory Stress: One potential way to increase L^{np} involves accessing large amounts of memory so that several page faults are generated in succession. The kernel invokes the page fault handler repeatedly and can thus execute long non-preemptable code sections.

Caps-Lock Stress: A quick inspection of the kernel code reveals that when the num-lock or caps-lock LED is switched, the keyboard driver sends a command to the keyboard controller and then spins while waiting for an acknowledgement interrupt. This process can potentially disable preemption for a long time.

Console-Switch Stress: The console driver code also seems to contain long non-preemptable paths that are triggered when switching virtual consoles.

I/O Stress: When the kernel or the drivers have to transfer chunks of data, they generally move this data inside non-preemptable sections. Hence, system calls that move large amounts of data from user space to kernel space (and vice-versa) and from kernel memory to a hardware peripheral, such as the disk, can cause large latencies.

Procfs Stress: Other potential latency problems in Linux are caused by the `/proc` file system. The `/proc` file system is a pseudo file system used by Linux to share data between the kernel and user programs. Concurrent accesses to the shared data structures in the `proc` file system must be protected by non-preemptable sections. Hence, we expect that reading large amounts of data from the `/proc` file system can increase the latency.

Fork Stress: The `fork()` system call can generate high latencies for two reasons. First, the new process is created inside a non-preemptable section and involves copying large amounts of data including page tables. Second, the overhead of the scheduler increases with increasing number of active processes in the system.

Experience and careful code analysis by various members of the Linux community (for example, see Senoner [18]) confirms that the above list of latency sources is comprehensive, i.e., it triggers a representative subset of long non-preemptable sections in the kernel and in the drivers.

4. Evaluation of the Kernel Latencies

In this section, we present an evaluation of the various OS latency components. We ran our experiments on a 1.8 Ghz Athlon processor with 512 MB of memory.

4.1. Timer Resolution Latency

The first set of experiments measures L^{timer} and shows that it can be easily eliminated from the OS non-preemptable section latency by using high resolution timers. We evaluated the high-resolution timer Linux kernel (standard kernel + our implementation of a high resolution timers mechanism) and compared its timer resolution latency with the timer latency of a standard Linux kernel. Consider the periodic task described in Section 3.1: In the standard kernel, if the task period is not a multiple of T^{tick} then the difference between the inter-activation times and T will be greater be 0 and can be as large as T^{tick}. As explained, this problem is solved by the high-resolution timer kernel, which we demonstrate through experiments described below.

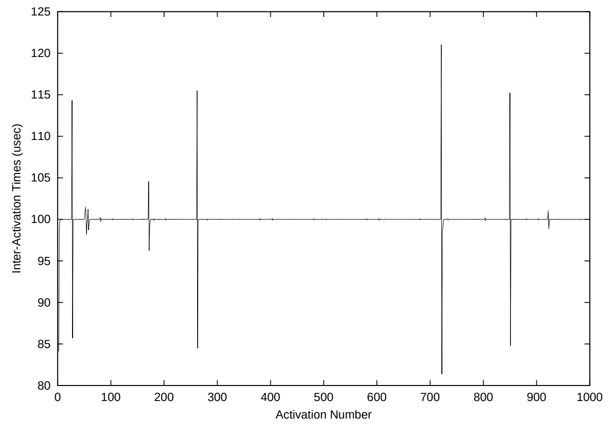

Figure 1. Inter-Activation times for a task that is woken up by a periodic signal with period $100\mu s$ on a high resolution timer Linux.

Figure 1 shows the inter-activation times measured with period $T = 100\mu s$ on the high-resolution timer kernel. Note that after 1000 activations the maximum difference between the period and the actual inter-activation time is less that $25\mu s$.

We repeated this experiment with different periods where each experiment was run for 10000000 activations. These new experiments showed that the difference between the period and the inter-activation time does not significantly depend on the period T. Figure 2 shows the Probability Distribution Function (PDF) of the inter-activation

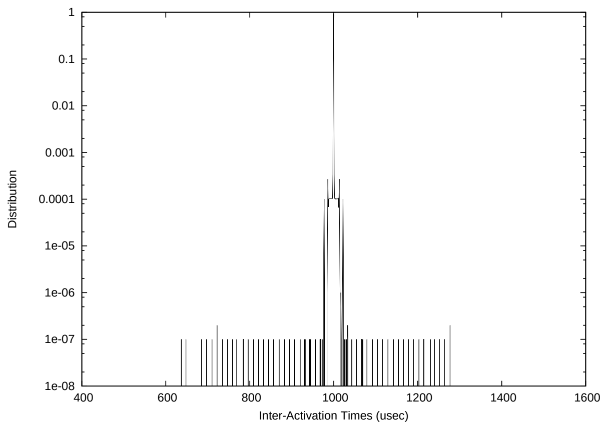

Figure 2. PDF of the difference between inter-activation times and period, when $T = 1000\mu s$.

times when $T = 1000\mu s$. The maximum measured inter-activation time is about $1300\mu s$, whereas the minimum is about $630\mu s$, and this distribution does not significantly vary with increasing number of activations.

We hypothesize that the maximum deviation between inter-activation times (about $370\mu s$) is due to the OS non-preemptable section latency L^{np}. However, we do not know the precise cause of this latency since we did not specifically control the background task set.

Hence, we performed a new set of experiments to measure latencies due to the various activities that can trigger long non-preemptable paths.

4.2. OS Non-Preemptable Section Latency

In this set of experiments, we used the `usleep()` test program described in Section 3.2 with $T = 100\mu s$ to measure and identify the causes of OS non-preemptable section latency. We performed these experiments on four different kernels described in Section 2.2.2: 1) the standard Linux kernel, 2) the Low-Latency Linux kernel, 3) the Preemptable Linux kernel, and 4) the Preemptable Lock-Breaking Linux kernel. Recall that the Low-Latency kernel uses Andrew Morton's Low-Latency patch [14] and the Preemptable Linux kernel and the Preemptable Lock-Breaking kernel use Robert Love's kernel preemption patch [11]. In the following, we will refer to those specific patches using the names presented above. The next section describes the initial set of experiments that we performed to understand which activities cause large OS non-preemptable section latencies. Section 4.2.2 describes additional experiments that we performed to test the sensitivity of the system to the order and the length of experimental runs.

4.2.1 Initial Analysis

The `usleep()` test program is started on an unloaded machine. Then the load-generating tasks described in Section 3.2 are run in the background to trigger long non-preemptable paths. To easily represent the latency results in a single plot per Linux variant, we used a background load that was generated as follows:

1. The memory stress test allocates a large integer array with a total size of 128 MB and accesses it sequentially. This test starts at $1000ms$, and finishes around $2000ms$.
2. The caps-lock stress test runs a program that switches the caps-lock LED twice. This test turns on the LED at $7000ms$ and then turns it off at $8000ms$.
3. The console-switch stress test runs a program that switches virtual consoles on Linux twice, first at $9000ms$ and then at $10000ms$.
4. The I/O stress test uses the `read()` and `write()` system calls and accesses 2 MB of data. This test starts at $11000ms$ and finishes around $13000ms$.
5. The procfs stress test reads a 512 MB file in the `/proc` file system. It runs from $17000ms$ to around $18000ms$.
6. The fork test forks 512 processes. This test starts at $20000ms$.

We ran the experiments on a standard Linux kernel and verified that the timer resolution latency L^{timer} is generally larger than L^{np} and hides the effects of disabling preemption for a long time. Hence, high resolution timers are needed to investigate L^{np}.

We repeated the experiment using high resolution timers in the `usleep()` implementation, and we obtained the results shown in Table 1. The table shows that the Low-Latency kernel can reduce the latency during the memory stress test and the I/O stress test, but does not reduce the latency generated by console switch, by caps-lock switch, and by the procfs stress. On the other hand, the Preemptable kernel can reduce the latency generated by the procfs stress, but re-introduces large latencies during the memory stress. Finally, the Lock-Breaking kernel seems to provide some of the benefits of the Low-Latency kernel (the latency during the memory stress is low) together with some of the benefits of the Preemptable kernel (for instance, the latency caused by the console switch test and by the procfs stress test). In summary, the largest latency is caused by the caps-lock stress test and all other latencies are within $1ms$.

Figure 3 shows a graphical representation of the results for the monolithic kernel with high resolution timers and provides further insight into the causes of latency (for the

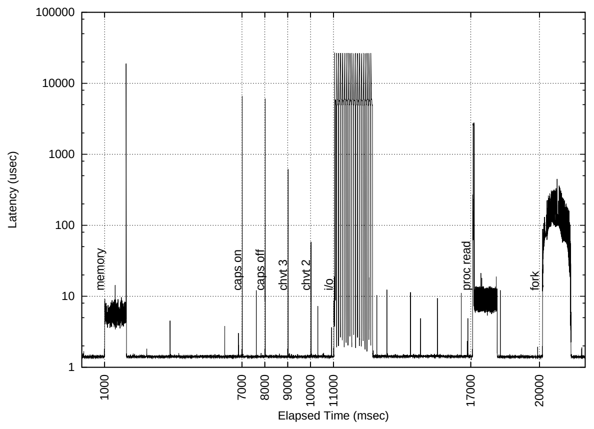

Figure 3. OS non-preemptable section latency measured on a high-resolution timer Linux. This test is performed with heavy background load.

sake of brevity, we omit the plots for other kernels, which are similar to this one). For instance, the figure shows that the big latency in the memory stress test that we see in Table 1 occurs only at the termination of the program. We found that the source of this latency is the `munmap()` system call which unmaps large memory buffers during program exit.

4.2.2 Sensitivity Analysis

For sensitivity analysis, we performed additional experiments by running the stress programs in several different orders and for a different lengths of time. The console switch and caps-lock tests did not show any difference with respect to the values in Table 1, thus confirming that: 1) none of the evaluated patches reduces the caps-lock switch latency, and 2) the Preemptable and Lock-Breaking kernels can reduce the console switch latency with respect to the standard or Low-Latency kernel.

Table 2 shows the maximum OS non-preemptable section latency measured when running the memory stress test, the I/O stress test, the procfs stress test and the fork stress test for a long time. The tests were run for 10 hours and 36000000 samples were collected. Although the worst case values shown in Table 2 are higher than in Table 1, the results are qualitatively similar. Thus, 1) the Low-Latency kernel reduces the latency during the memory stress test but not during the procfs stress or during console switch tests, 2) the Preemptable kernel reduces the latency during the procfs stress and during console switch tests but not during the memory test, and 3) the Lock-Breaking kernel reduces all these latencies.

Figure 4 shows the Cumulative Distribution Function

	Memory Stress	Caps-Lock Caps-Lock	Console Switch	I/O Stress	Procfs Stress	Fork Stress
Monolithic	18212	6487	614	27596	3084	295
Low-Latency	63	6831	686	38	2904	332
Preemptable	17467	6912	213	187	31	329
Preemptable Lock-Breaking	54	6525	207	162	24	314

Table 1. OS non-preemptable section latencies (in μs) for different kernels under different loads (test run for 25 seconds).

	Memory Stress	I/O Stress	ProcFS Stress	Fork Stress
Monolithic	18956	28314	3563	617
Low-Latency	293	292	3379	596
Preemptable	18848	392	224	645
Preemptable Lock-Breaking	239	322	231	537

Table 2. OS non-preemptable section latencies (in μs) for different kernels under different loads (tests run for 10 hours).

(CDF) $P\{L^{np} < l\}$ of the OS non-preemptable section latencies measured during the I/O stress test. Note that for all these three kernels the probability of measuring latencies higher than $20\mu sec$ is less than 0.01. The graph shows that that the Preemptable and Lock-Breaking kernels have lower latency with higher probability (the CDF increases faster). For example, $P\{L^{np} < 10\mu s\}$ is 0.99466 on a Preemptable kernel, 0.99541 on a Lock-Breaking kernel, and 0.441798 on a low-latency kernel. However, while not visible in Figure 4, these kernels have latency distributions with longer tails: for example, $P\{L^{np} < 40\mu s\}$ is 0.997099 on a Preemptable kernel, $0.0.998165$ on a Lock-Breaking kernel, and 0.999874 on a low-latency kernel. In this sense, the Low-Latency kernel provides better real-time performance.

5. Effects on a Real Application

In this section, we examine the effects of the OS latency on a real Linux application. As a test application, we selected *mplayer* [1], an audio/video player that can handle several different media formats.

Mplayer synchronizes audio and video streams by using timestamps that are associated with the audio and video frames. The audio card is used as a timing source, i.e., audio samples are put in the audio card buffer, and when a video frame is decoded, its timestamp is compared with the timestamp of the currently played audio sample. If the video timestamp is smaller than the audio timestamp then the program is late (i.e., a video deadline has been missed) and the video is immediately displayed. Otherwise, the system sleeps until the video timestamp and the audio timestamp are equal and then displays the video.

Assuming no OS latency and a fast enough CPU, audio/video synchronization can be achieved by simply sleeping for the correct amount of time (and in fact mplayer sleeps using the Linux `usleep()` call). Unfortunately, if the OS latency is high, mplayer will not be able to sleep for the correct amount of time leading to poor audio/video synchronization.

To verify this hypothesis, we instrumented mplayer to measure the time when a video frame is displayed and the difference between the audio and video timestamps at display time. Using this instrumented version of mplayer, we performed some experiments on a standard Linux kernel (high latency) and on a lock-breaking preemptable Linux kernel with high resolution timers (reduced latency). To show the effects of the OS latency, we ran the I/O stress test as a competing load while running mplayer at the highest real-time priority. This test spends about 90% of its execution time in kernel mode. As described in Section 4.2, the I/O stress test performs intensive file system accesses and exacerbates the kernel preemptability problem.

Figure 5 shows the difference between the audio and the video timestamps when the video frame is displayed for mplayer running on standard Linux. On standard Linux, the maximum difference between audio and video timestamps is more than $80000us$, and the figure qualitatively shows there is a large variance in this difference. Note that the audio/video skew in mplayer can be negative (by as much as $5ms$) due to the $10ms$ resolution of the kernel timers.

Figure 6 presents the results obtained using the lock-breaking preemptable Linux kernel with high resolution

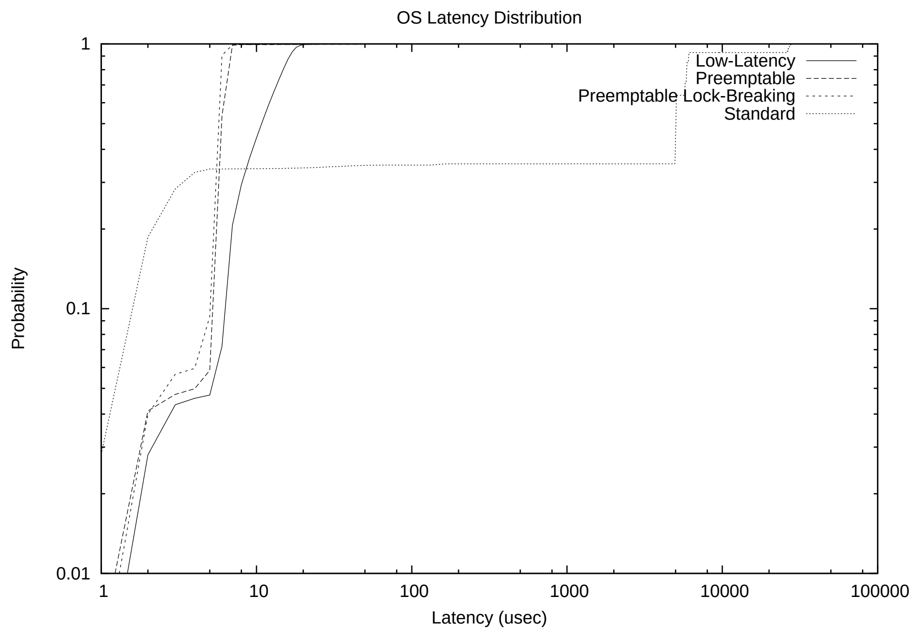

Figure 4. CDF of the latency measured on different versions of Linux (with high resolution timers). This test is performed with the I/O stress in background.

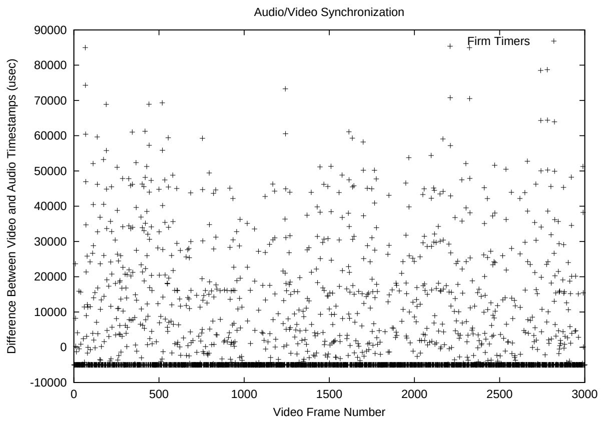

Figure 5. Audio/Video Skew on standard Linux. Heavy kernel load is run in the background.

timers. In this case, the difference between audio and video timestamps is significantly lower and the maximum difference is less than $400\mu s$.

The second set of results show the *inter-frame times*, i.e. the difference between the display times of a video frame and the previous frame. The expected inter-frame time is the process period $1/F$ where F is the video frame rate. In our experiments, we used an MPEG movie with a video frame rate of 30 frames per second. Thus the expected inter-frame time is $33.3ms$. Figure 7 shows the inter-frame times obtained using standard Linux. Since L^{timer} can be up to $10ms$, we expect the inter-frame times to cluster around $30ms$ and $40ms$. However, the L^{np} component due to the background load introduces additional variation in the inter-frame times and increases these times to more than $100ms$ (or $100000\mu s$).

In contrast, Figure 8 shows the inter-frame times obtained using the lock-breaking preemptable kernel with high resolution timers. The inter-frame times are clustered around the correct value of $33.3ms$ and their variation is very low.

6. Related Work

Although we are not aware of any previous systematic study of the Linux latency, some of the issues highlighted in this paper have been addressed in the past during the design of real-time operating systems and real-time extensions to Linux.

In particular, many different real-time algorithms have been implemented in Linux and in other general purpose kernels. For example, Linux/RK [15] implements Resource Reservations in the Linux kernel, and RED Linux [26] provides a generic scheduling framework for implement-

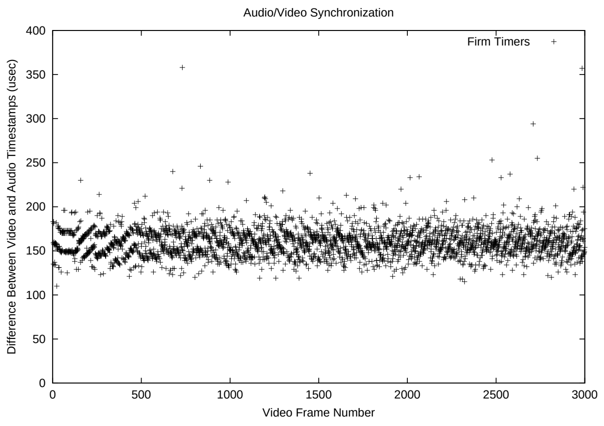

Figure 6. Audio/Video Skew for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background. The Audio/Video skew is clustered around 0, and the maximum skew is less than $400us$ (note that the scale is different from Figure 5).

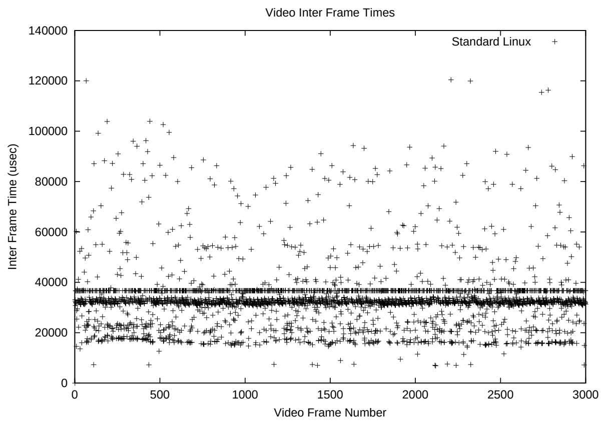

Figure 7. Inter-Frame times for standard Linux. Heavy kernel load is run in the background.

ing different real-time scheduling algorithms. Several proportional share scheduling mechanisms have been implemented [20, 22, 7, 25] in the FreeBSD, Linux, or Solaris kernels and DSRT [8] is a user-level scheduling solution.

While implementing real-time scheduling in general purpose kernels, the authors of the previous work noticed the latency problems, and some of the previous systems address them. For example, RED Linux inserts preemption points in the kernel (transforming it to a Low-Latency kernel), and Timesys Linux/RT (based on RK technology) uses full kernel preemptability for reducing the OS latency. Kernel preemptability is also used by MontaVista Linux [23] whose preemptable kernel patch has been recently accepted in the 2.5.4 kernel. It is worth noting that the advantages of a preemptable kernel were already well known in the real-time community [13]. Recently, there has been renewed interest in the evaluation of these latency-reduction techniques. Concurrent with our work (and unknown to us), Clark Williams from Red Hat [24] evaluated Linux scheduling latency in a manner similar to the one presented in this paper. The main difference is that Williams uses a decomposition of the OS latency that is different from ours and he does not explicitly consider timer resolution latency. Williams comes to similar conclusions as us although his numerical results are slightly different from our results. One probable reason for this discrepancy is that he uses a different version of the kernel patches and he did not use the lock-breaking patch. We are still investigating how his numbers relate to our results.

A different approach for reducing the OS latency is used by other systems, such as RTLinux [5], RTAI [12], and KURT [19], which decrease the latency by running Linux as a background process over a small real-time executive. In this case, real-time tasks are not Linux processes, but run on the lower-level real-time executive, and the Linux kernel runs as a non real-time task. This solution provides good real-time performance to the real-time tasks executing in kernel space, but does not provide it to standard Linux applications. Linux processes are still non real-time, hence we believe that RTLinux-like solutions are not usable for supporting time-sensitive applications running in user space.

As shown in Section 4, the latency L^{timer} due to the timer resolution can be eliminated by using high resolution timers. For this reason, most of the existing real-time kernels or real-time extensions to Linux provide high resolution timers. The high resolution timers concept was proposed by RT-Mach [17] and has subsequently been used by Rialto [10], RED Linux [26], RTLinux [5], and Linux/RK [15] just to cite some examples. Moreover, MontaVista provides a patch for the standard Linux kernel implementing high resolution timers [4].

7. Conclusions and Future Work

In this paper, we have evaluated the real-time behavior of Linux by measuring the *latency* of various kernel variants. This evaluation is important because it enables the application of real-time guarantees to the Linux system, where latencies are modeled as blocking times.

In the future, we plan on using preemptable lock-breaking Linux (with a high resolution timers mechanism) to implement a reservation-based system that provides pre-

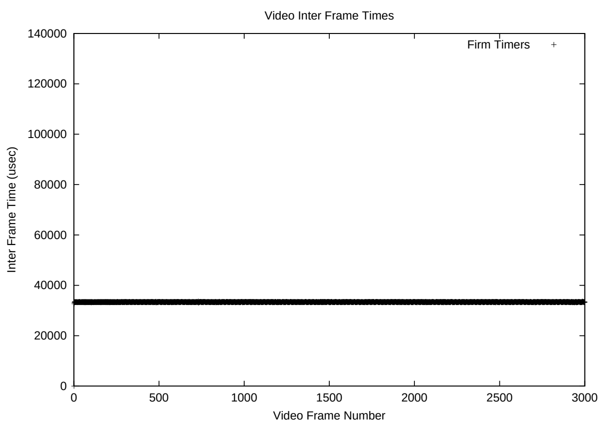

Figure 8. Inter-Frame times for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background.

dictable scheduling. In addition, we plan to extend our analysis of OS latency to separate the kernel non-preemptable sections latency from interrupt processing overhead. For example, in Linux, an intensive interrupt load can cause long OS latencies due to the design of the interrupt processing mechanism (ISRs, tasklets, and bottom halves). Preliminary results show that the effects of interrupt processing can be mitigated by using resource reservations together with some adaptation strategy [16, 3, 2].

References

[1] Mplayer - movie player for linux. http://www.mplayerhq.hu.

[2] L. Abeni. Coping with interrupt execution time in real-time kernels: a non-intrusive approach. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, London, UK, December 2001.

[3] L. Abeni and G. Lipari. Compensating for interrupt process times in real-time multimedia systems. In *Third Real-Time Linux Workshop*, Milano, Italy, November 2001.

[4] G. Anzinger. High resolution timers project. http://high-res-timers.sourceforge.net/.

[5] M. Barabanov and V. Yodaiken. Real-time linux. *Linux Journal*, March 1996.

[6] F. S. Foundation. About free software. http://www.gnu.org/philosophy/.

[7] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of the 2nd OSDI Symposium*, October 1996.

[8] H. hua Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[9] T. Inc. Timesys linux. http://www.timesys.com.

[10] M. B. Jones, J. S. B. III, A. Forin, P. J. Leach, D. Rosu, and M.-C. Rosu. An overview of the rialto real-time architecture. In *In Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[11] R. Love. The linux kernel preemption project. http://kpreempt.sourceforge.net/.

[12] P. Mantegazza, E. Bianchi, L. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 72, 2000.

[13] C. W. Mercer and H. Tokuda. Preemptibility in real-time operating systems. In *In Proceedings of the 13th IEEE Real-Time Systems Symposium*, December 1992.

[14] A. Morton. Linux scheduling latency. http://www.zip.com.au/ akpm/linux/schedlat.html.

[15] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

[16] J. Regehr and J. 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.

[17] S. Savage and H. Tokuda. Rt-mach timers: Exporting time to the user. In *In Proceedings of USENIX 3rd Mach Symposium*, April 1993.

[18] B. Senoner. Audio latency benchmark. http://www.gardena.net/benno/linux/audio/.

[19] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1998.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[21] L. Torvalds et al. The linux kernel. http://www.kernel.org.

[22] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating System Design and Implementation*, pages 1–12, November 1994.

[23] B. Weinberg and C. Lundholm. Embedded linux - ready for real-time. In *Third Real-Time Linux Workshop*, Milano, Italy, November 2001.

[24] C. Williams. Linux scheduler latency. http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf, Mar 2002.

[25] D. K. Y. Yau and S. S. Lam. Adaptive rate controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, August 1997.

[26] Yu-Chung and K.-J. Lin. Enhancing the Real-Time Capability of the Linux Kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.