A Measurement-Based Analysis of the Real-Time Performance of the Linux Kernel

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 kernel. 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 kernel: 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.

Paper Category: Research Paper

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).

*Luca Abeni is also affiliated with ReTiS Lab, Scuola Superiore S. Anna, Pisa, Italy.

In particular, we define a metric called *kernel 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 the Linux kernel [22]. We chose Linux because it is widely used, supports most commonly available hardware and is distributed under an open source license [7] which enables researchers to easily experiment with it.

We identify several sources of kernel latency, the two most important sources being *timer resolution* and *non-preemptive sections* in the kernel. We designed a set of micro-benchmarks and have used these benchmarks to systematically quantify each source of latency in the Linux kernel. 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 kernel 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 kernel latency and investigates the factors that contribute to it. In Section 3, we describe the experimental setup for evaluating the various components of kernel latency. Section 4 presents the experimental results, and in Section 5 we show how kernel 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 Kernel Latency

The main focus of the Linux kernel is to provide high throughput, i.e., minimizing the average execution time experienced by concurrently executing processes. Until now, 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 *kernel latency* as a metric to evaluate kernel support for time-sensitive applications. We define kernel 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 kernel 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 Kernel Latency

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

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.

Non-preemptable section latency is caused by non-preemptable sections in the kernel that can keep a high priority task from being scheduled. For example, if 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 another task is running in the kernel in a non-preemptable section. In this case, τ will be scheduled when the kernel exits the non-preemptable section at time t', contributing a *non-preemptable section* latency $L^{np} = t' - t$.

[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.

2.2 Analysis of the Latencies

In our experiments, τ is scheduled using the highest real-time priority to eliminate kernel 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[18]. In an x86 architecture, the PIT or the 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 4.2.1 show that the resolution of our high-resolution timers lies between $4\mu s$ to $6\mu s$.

2.2.2 Non-Preemptable Sections

The second term contributing to the maximum kernel latency is the maximum size of a kernel non-preemptable section, $\max\{L^{np}\}$. This value depends 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 kernel latencies due to non-preemptable sections of 4 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 $100ms$ 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 [16], and by Andrew Morton's low-latency patch [15]. 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 [12] 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.

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.

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] There is also a different patch, from Timesys [10], based on mutexes and priority inheritance instead of on spinlocks.

3 Experimental Setup

The goal of this paper is to evaluate Linux kernel 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 non-preemptable section latency L^{np} in isolation. To measure L^{timer}, we eliminate L^{np} by running the experiment on an unloaded 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 latency due to non-preemptable sections 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 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 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 a non-preemptive section of size S, the latency must be sampled with a period $T \ll S$ or else the non-preemptive code could execute between two consecutive measurements. 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 $L^{np} = t_2 - (t_1 + T)$

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 (as explained in Section 2).

Memory Stress: One potential way to increase kernel latency 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 generate long non-preemptable latencies.

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 has to transfer chunks of data, it generally moves 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 non-preemptable 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 kernel latency.

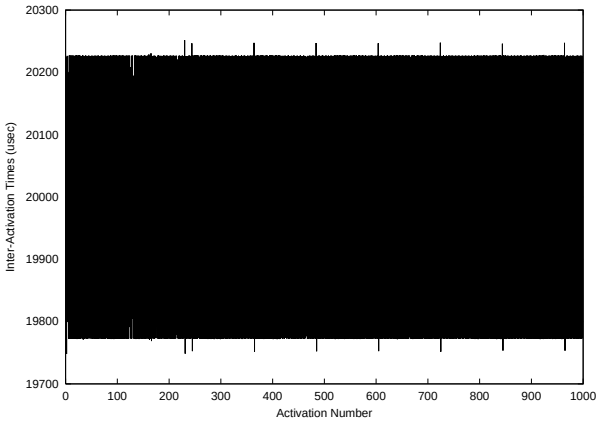

Figure 1: Inter-Activation times for a task that is woken up by a periodic signal with period $11ms$ on a standard Linux kernel. Note that the task period is greater than $T^{tick} = 10ms$.

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 [19]) confirms that the above list of latency sources is comprehensive, i.e., it triggers a representative subset of long non-preemptable sections in the Linux kernel.

4 Evaluation of the Kernel Latencies

In this section, we present an evaluation of the various kernel latency components. We ran our experiments on a 450 Mhz Pentium II 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 kernel latency by using high resolution timers. We ran these experiments using a periodic task as described in Section 3. We compared two different kernels, the standard Linux kernel and the high-resolution timer Linux kernel, as described in Section 2.2.1.

Figure 1 shows the inter-activation times on a standard Linux kernel when $T = 11ms$. We expect that if the task period is not a multiple of T^{tick} then the difference between the inter-activation times and T is not 0 but close to $T\%T^{tick}$. In the figure above, we notice as expected that this difference is

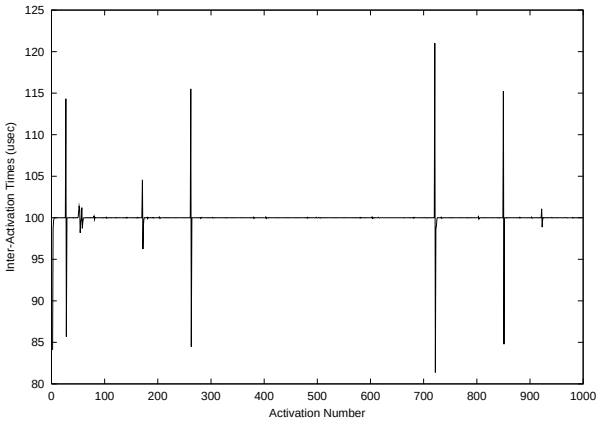

Figure 2: Inter-Activation times for a task that is woken up by a periodic signal with period $100\mu s$ on a high resolution timer Linux. Note that the task period is much smaller than in Figure 1.

$T(\mu s)$	100	200	300	400	500	600	700	800	900
$L(\mu s)$	47	51	43	44	49	53	50	52	50
$T(\mu s)$	1000	2000	3000	4000	5000	6000	7000	8000	9000
$L(\mu s)$	46	47	52	48	51	49	55	50	57
$T(\mu s)$	10000	20000	30000	40000	50000	60000	70000	80000	90000
$L(\mu s)$	52	46	51	49	54	50	43	47	51

Table 1: The table shows L, the maximum difference between the inter-activation times and the task period, for different values of the task period T on a high resolution timer Linux.

close to $9ms$.

Next, we repeated the experiments using the high-resolution timer kernel. As expected, high resolution timers greatly reduce latency. For example, Figure 2 shows the inter-activation times measured for $T = 100\mu s$. Note that this period results in a much tighter latency requirement as compared to the one in Figure 1 and that after 1000 activations the maximum difference between the period and the actual inter-activation time is less that $25\mu s$. Hence, we conjecture that the $9ms$ latency shown in Figure 1 is almost completely due to the timer resolution.

Table 1 shows the maximum absolute value of the difference between the period and the inter-activation times for various values of T on a high resolution timer kernel. Each of these maximum values has been measured over $1,000,000$ activations. The table shows that the maximum difference does not significantly depend on the period T and its maximum value is about $57\mu s$. We hypothesize that this value is due to the 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 the latencies due to the various activities

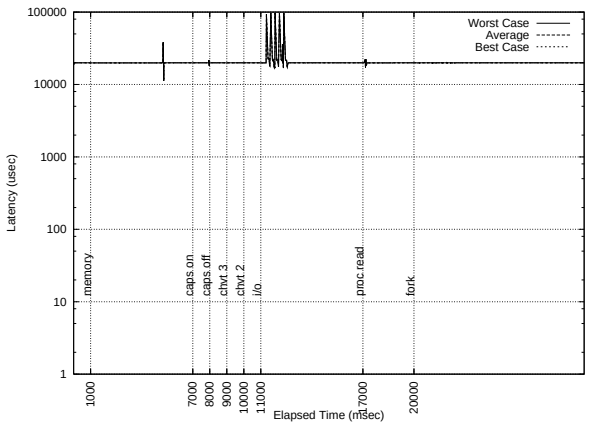

Figure 3: Latency measured on a standard Linux kernel. This test is performed with background load. Note that the L^{timer} component dominates the latency.

that can trigger long non-preemptable paths.

4.2 Non-Preemptable Section Latency

In this set of experiments, we used the `usleep()` test program described in Section 3 with $T = 100\mu s$ to measure and identify the causes non-preemptable section latency. This test program was started on an unloaded machine [4] and then the load-generating tasks described in Section 3.2 were run sequentially in the background to trigger long non-preemptable paths. We performed these experiments on four different kernels: 1) 1) the standard Linux kernel, 2) the Low-Latency Linux kernel, 3) the Preemptable Linux kernel, and 4) the Preemptable Lock-Breaking Linux kernel.

The background load is generated in the following way:

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 $5000ms$.

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 $13000ms$.

[4] The tests were performed in single user mode to minimize the number of system activities.

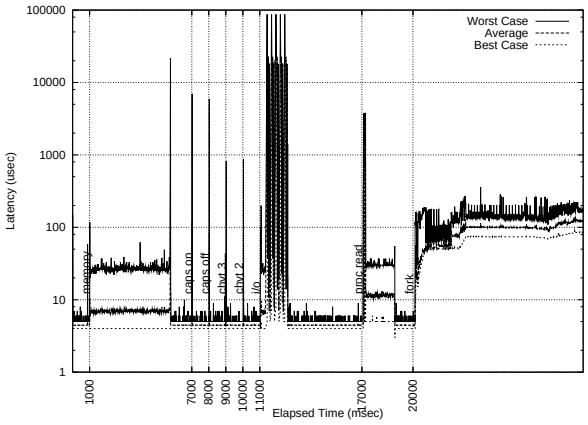

Figure 4: Latency measured on a Linux kernel with high resolution timers. This test is performed with background load. Now, L^{np} is visible.

5. The procfs stress test reads a 512 MB file in the `/proc` file system. It runs from 17000ms to around 19000ms.

6. The fork test forks 512 processes. This test starts at 20000ms and continues until the end of the experiment.

4.2.1 Standard Linux

Figure 3 shows the latency measured on a standard (monolithic) Linux kernel (version 2.4.16). Due to the implementation of the `usleep()` call on Linux, L^{timer} is around 19.9ms instead of 9.9ms. The memory access test, starting at $t = 1000ms$ does not seem to create any additional latency. However, we notice a small spike at the end of the test around $t = 5000ms$ (explained in the next experiment). In this experiment, we do not notice any variation in the latency during the caps-lock stress test or the console-switch test. On the other hand, there are some large spikes (up to 100ms) from $t = 11000ms$ to $t = 13000ms$ during the the I/O stress test. Note that the Y axis is shown on a logarithmic scale. None of the other tests present any significant contribution to kernel latency. Hence, we conclude that in a standard Linux kernel, the timer resolution latency L^{timer} is generally larger than L^{np} and hides the effects of non-preemptable sections. We believe that this is one reason why latency problems have not been previously addressed by the Linux community. These results show that we need to use a high resolution timers mechanism to investigate L^{np}.

Figure 4 reports the results obtained when high resolution timers are used in the `usleep()` implementation. It shows that in this case L^{timer} is almost completely removed. Hence, the effects of long non-preemptable sections are more visible. For instance, when the system is unloaded ($t < 1000ms$)

the latency lies between $4\mu s$ and $6\mu s$. This latency is due to the resolution of the timing mechanism and it matches the expected value of the interrupt service time on the Pentium II processor. It increases to $20\mu s$ during the memory stress test. This result is surprising because contrary to common belief it shows that page faults of other processes in Linux are not a serious problem for real-time performance. However, the end of the memory stress test generates a spike of about $20ms$ in kernel latency. We found that the source of this latency is the `munmap()` system call when large memory buffers are unmapped.

The caps-lock shift significantly increases kernel latency. During the caps-lock stress test ($t = 7000ms$ and $t = 8000ms$) the latency rises to $7ms$. On the other hand, the console switch test ($t = 9000ms$ and $t = 10000ms$) only increases the latency to $900\mu s$. Again, the longest critical paths seem to be triggered by the I/O stress test between $t = 11000ms$ and $t = 13000ms$ when the latency increases to $100ms$, similar to the previous experiment. Finally, the procfs stress test can contribute about $4ms$ to latency, whereas the fork test contributes up to about $300\mu s$. Again, note that in a standard Linux kernel, the $10ms$ resolution of the timers hides most of these values except the latency caused by file accesses.

From Figure 4, we expect reduction in the latency if the length or granularity of the kernel non-preemptable sections is reduced. As explained in Section 2, there are several ways in which non-preemptable kernel sections can be shortened. First, preemption points can be manually placed to break long non-preemptable paths, such as in the low-latency kernel. Second, the kernel can be made fully preemptable, where preemption is disabled only when spinlocks are held. Finally, the first technique can be used to reduce the length of spinlocks in a preemptable kernel. These techniques are explored in the next sections.

4.2.2 Low-Latency Linux

We evaluated Andrew Morton's patch [15] as an example of a low-latency kernel. Figure 5 shows the latency measured on a high resolution timers kernel with the Andrew Morton low-latency patch.

First, we note that the latency experienced during the memory stress test does not change significantly, but the $20ms$ spike caused by unmapping the large memory buffer has been removed. Now the `munmap()` latency is about $200\mu s$. However, the latency caused by the caps-lock and console stress tests is not changed, and in this experiment the worst latency is caused by toggling the caps-lock key! We see that the latency spikes between $t = 11000ms$ and $t = 13000ms$ have disappeared and thus the I/O stress test does not cause serious problems for real-time performance anymore. However, the latency caused by the procfs stress test and by the fork stress test is unchanged as compared to the

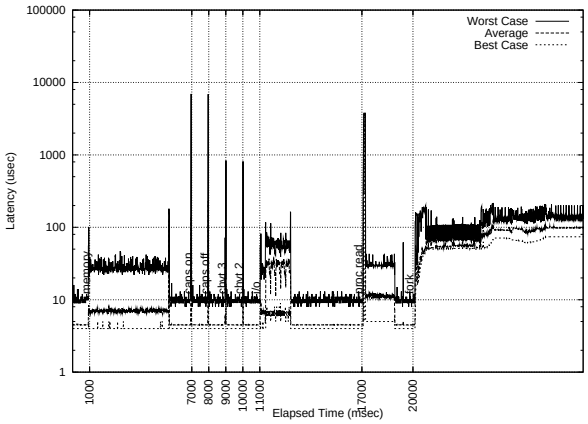

Figure 5: Latency measured on a Low-Latency Linux kernel with high resolution timers. The `munmap()` and I/O latencies are reduced.

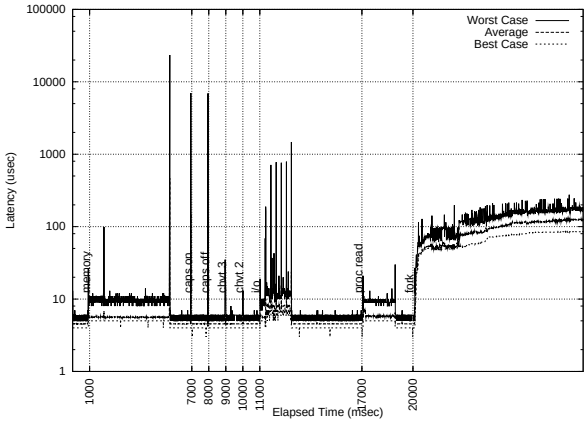

Figure 6: Latency measured on a Preemptable-Linux kernel with high resolution timers. The `procfs` latency is reduced, but the `munmap()` latency becomes high again.

monolithic kernel.

In summary, the latency caused by all the activities except the procfs stress test and the caps-lock stress test is under $1ms$.

4.2.3 Preemptable Linux

Figure 6 shows the results obtained using a Preemptable-Linux kernel [12]. The big difference we notice as compared to the Low-Latency kernel is that the `munmap()` system call causes high latency once again (about $20ms$ around time $t = 5000ms$). The latency caused by the I/O stress test is also increased with spikes up to $1ms$. On the other hand, the procfs stress test does not cause significant latency. In particular, the big spike in latency at time $t = 17000ms$ has been removed. In this

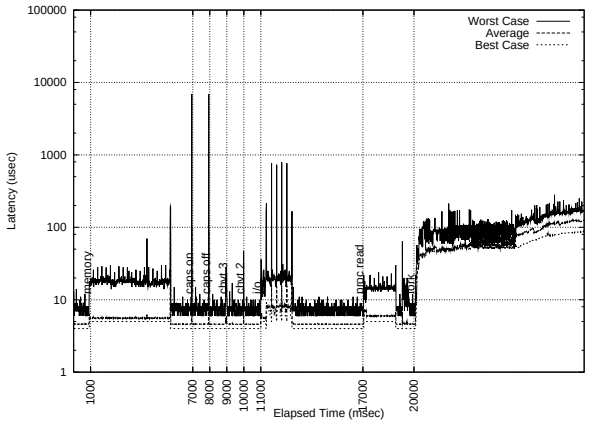

Figure 7: Latency measured on a Lock-Breaking Preemptable-Linux kernel with high resolution timers. Note that most of the latencies are under $1ms$.

experiment, the worst latency is caused by the `munmap()` system call and is due to the kernel holding a spinlock for a long time. We will see that the lock-breaking approach described below will solve this problem.

4.2.4 Lock-Breaking Preemptable Linux

Figure 7 shows the results obtained when the lock-breaking preemptable kernel is used. We see that breaking long spinlocks solves the `munmap()` problem. The kernel behavior during the memory stress (and during the final `unmap()`) is similar to the behavior of the low-latency kernel. Moreover, this kernel also has the benefits of the preemptive kernel. For instance, compared to the low-latency kernel, there are improvements in the latency caused by the console switch stress 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 $1ms$.[5] Also, we would like to stress that in the previous experiments we have tried to generate the "worst case" kernel latency. We do not expect that latencies larger than the ones reported can be easily generated.

5 Effects on a Real Application

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

[5] File accesses are still not as low as in Figure 5. We suspect this latency is caused by heavy interrupt loads and long non-preemptable interrupt processing times. We plan to study this issue further in the future.

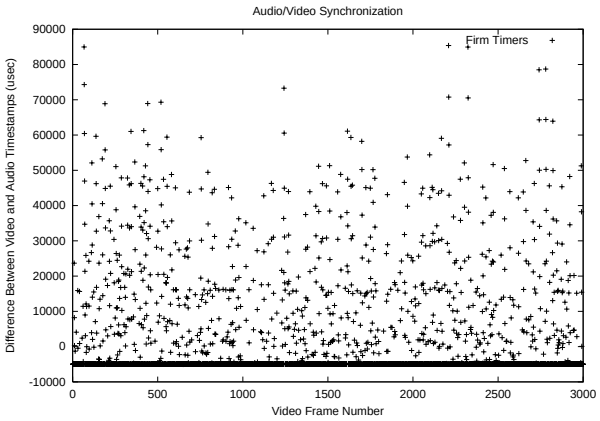

Figure 8: Audio/Video Skew on standard Linux. Heavy kernel load is run in the background.

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 kernel 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 kernel 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 kernel latency) and on a lock-breaking preemptable Linux kernel with high resolution timers (reduced kernel latency). To show the effects of kernel 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 disk accesses and exacerbates the kernel preemptability problem.

Figure 8 shows the difference between the audio and the video timestamps when the video frame is displayed for mplayer running on the Standard Linux kernel. On standard Linux, the maximum

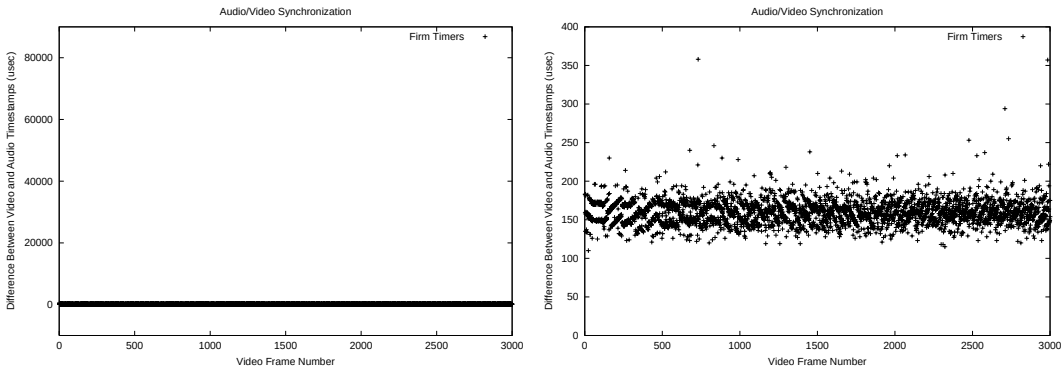

Figure 9: Audio/Video Skew for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background. Figure a) shows that the Audio/Video skew is clustered around 0, and Figure b) zooms the plot to show that the maximum skew is less than $400us$.

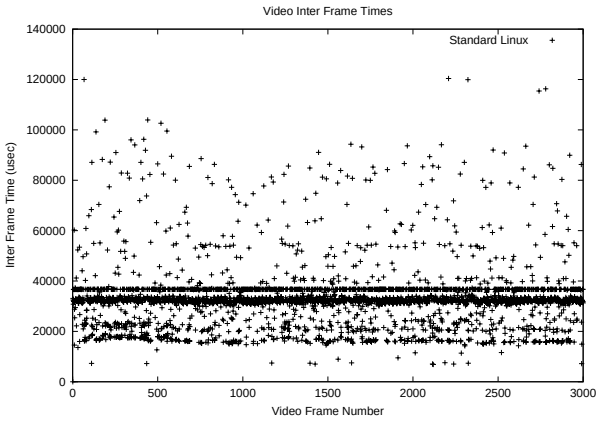

Figure 10: Inter-Frame times for standard Linux. Heavy kernel load is run in the background.

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

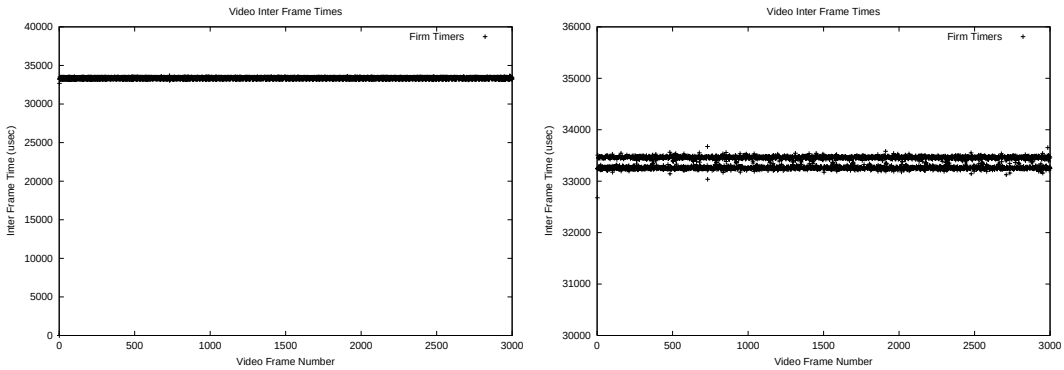

Figure 11: Inter-Frame times for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background. Figure a) shows that inter-frame times are clustered around $33.3ms$, and Figure b) zooms the plot to show that the inter-frame times lie between a $1ms$ interval. We are not sure why there are two clustered lines of inter-frame times. We plan to investigate this issue.

rate of 30 frames per second. Thus the expected inter-frame time is $33.3ms$. Figure 10 shows the inter-frame times obtained using the standard Linux kernel. 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 $100000us$).

In contrast, Figure 11 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.

These experiments show that the Linux kernel latency can be sufficiently reduced to support time-sensitive applications, such as multimedia applications, that have have latency requirements of $1ms$ or more.

6 Related Work

Although we are not aware of any previous systematic study of the Linux kernel 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 [16] implements Resource Reservations in the Linux kernel, and RED Linux [26] provides a generic scheduling framework for implementing different real-

time scheduling algorithms. Severl proportional share scheduling mechanisms have been implemented [21, 23, 8, 25] in the FreeBSD, Linux, or Solaris kernels and DSRT [9] 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 kernel latency. Kernel preemptability is also used by MontaVista Linux [24] 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 [14].

A different approach for reducing latency is used by other systems, such as RTLinux [6], RTAI [13], and KURT [20], 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, but does not provide it to Linux applications. Linux processes are still non real-time, and cannot support time-sensitive applications. Also, native real-time threads use a completely different, and less evolved, ABI compared to the Linux one, and do not have access to Linux device drivers.

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 [18] and has subsequently been used by Rialto [11], RED Linux [26], RTLinux [6], and Linux/RK [16] just to cite some examples. Moreover, MontaVista provides a patch for the standard Linux kernel implementing high resolution timers [4].

A problem that has been observed with high resolution timers is that they can lead to high interrupt processing overhead when used frequently. This problem is solved by the soft timer mechanism [5] and in our high-resolution timer implementation.

7 Conclusions and Future Work

In this paper, we have evaluated the real-time behavior of Linux by measuring the *kernel latency* of various Linux 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.

We identified the two most important sources of kernel latency: *timer resolution* and *non-preemptable sections* inside the kernel. Our experiments showed that in a standard Linux kernel the timer resolu-

tion latency is predominant (about 10 ms), and generally hides the non-preemptable section latency. However, when a high-resolution timer mechanism is implemented in the kernel, the latency due to kernel non-preemptable sections becomes visible. We compared the non-preemptable section latency of four readily available Linux variants: a standard (non-preemptable) kernel, a low-latency kernel, a preemptable kernel and preemptable lock-breaking kernel. Our results show that the preemptable lock-breaking kernel has the lowest overall kernel latency (ignoring the caps-lock problem, the worst case is less than 1 ms) and can thus provide reasonable real-time performance. Finally, we evaluated the effects of reduced kernel latency on a real application, and showed that it can significantly improve real-time performance.

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 predictable scheduling. In addition, we plan to extend our analysis of non-preemptable latency to include interrupt processing overhead. For example, in the Linux kernel, an intensive interrupt load can cause long non-preemptable latencies due to the design of the interrupt processing mechanism. Preliminary results show that the effects of interrupt processing can be mitigated by using resource reservations together with some adaptation strategy [17, 3, 2].

References

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

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

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

[5] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *In ACM Transactions on Computer Systems (TOCS)*, August 2000.

[6] Michael Barabanov and Victor Yodaiken. Real-time linux. *Linux Journal*, March 1996.

[7] Free Software Foundation. About free software. http://www.gnu.org/philosophy/.

[8] Pawan Goyal, Xingang Guo, and Harrik M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of the 2nd OSDI Symposium*, October 1996.

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

[10] TimeSys Inc. TimeSys Linux. `http://www.timesys.com`.

[11] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Rosu, and Marcel-Catalin Rosu. An overview of the rialto real-time architecture. In *In Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[12] Robert Love. The Linux kernel preemption project. `http://kpreempt.sourceforge.net/`.

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

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

[15] Andrew Morton. Linux scheduling latency. `http://www.zip.com.au/~akpm/linux/schedlat.html`.

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

[17] 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.

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

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

[20] 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.

[21] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[22] Linus Torvalds et al. The linux kernel. http://www.kernel.org.

[23] 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.

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

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

[26] Yu-Chung and Kwei-Jay 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.