Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes

Yoav Etsion* Dan Tsafrir Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
{etsman,dants,feit}@cs.huji.ac.il

ABSTRACT

It is commonly agreed that scheduling mechanisms in general purpose operating systems do not provide adequate support for modern interactive applications, notably multimedia applications. The common solution to this problem is to devise specialized scheduling mechanisms that take the specific needs of such applications into account. A much simpler alternative is to better tune existing systems. In particular, we show that conventional scheduling algorithms typically only have little and possibly misleading information regarding the CPU usage of processes, because increasing CPU rates have caused the common 100 Hz clock interrupt rate to be coarser than most application time quanta. We therefore conduct an experimental analysis of what happens if this rate is significantly increased. Results indicate that much higher clock interrupt rates are possible with acceptable overheads, and lead to much better information. In addition we show that increasing the clock rate can provide a measure of support for soft real-time requirements, even when using a general-purpose operating system. For example, we achieve a sub-millisecond latency under heavily loaded conditions.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling; D.4.8 [**Performance**]: Measurements; C.4 [**Performance of Systems**]: Design studies

General Terms

Measurement, Performance

Keywords

Clock interrupt rate, Interactive process, Linux, Overhead, Scheduling, Soft real-time, Tuning

*Supported by a Usenix scholastic grant.

1. INTRODUCTION

Contemporary computer workloads, especially on the desktop, contain a significant multimedia component: playing of music and sound effects, displaying video clips and animations, etc. These workloads are not well supported by conventional operating system schedulers, which prioritize processes according to recent CPU usage [18]. This deficiency is often attributed to the lack of specific support for real-time features, and to the fact that multimedia applications consume significant CPU resources themselves.

The common solution to this problem has been to design specialized programming APIs that enable applications to request special treatment, and schedulers that respect these requests [19, 8, 22]. For example, applications may be required to specify timing constraints such as deadlines. To support such deadlines, the conventional operating system scheduler has to be modified, or a real-time system can be used.

While this approach solves the problem, it suffers from two drawbacks. One is price. Real-time operating systems are much more expensive than commodity desktop operating systems like Linux or Windows. The price reflects the difficulty of implementing industrial strength real-time scheduling. This difficulty, and the requirement for careful testing of all important scenarios, are the reasons that many interesting proposals made in academia do not make it into production systems. The other drawback is the need for specialized interfaces, that may reduce the portability of applications, and require a larger learning and coding effort.

An alternative is to stick with commodity desktop operating systems, and tune them to better support modern workloads. While this may lead to sub-optimal results, it has the important benefit of being immediately applicable to the vast majority of systems installed around the world. It is therefore worth while to perform a detailed analysis of this approach, including what can be done, what results may be expected, and what are its inherent limitations.

1.1 Commodity Scheduling Algorithms

Prevalent commodity systems (as opposed to research systems) use a simple scheduler that has not changed much in 30 years. The basic idea is that processes are run in priority order. Priority has a static component (e.g. operating system processes have a higher initial priority than user processes) and a dynamic part. The dynamic part depends on CPU usage: the more CPU cycles used by a process, the lower its priority becomes. This negative feedback (running reduces your priority to run more) ensures that all processes get a fair share of the resources. CPU usage is forgotten after some time, in order to focus on recent activity and not on distant history.

While the basic ideas are the same, specific systems employ different variations. For example, in Solaris priorities of processes that wake up after waiting for an event are set according to a table, and the allocated quantum duration is longer if the priority is lower [17]. In Linux the relationship goes the other way, with the same number serving as both the allocation and the priority [5, 4]. In Windows NT and 2000, the priority and quanta allocated to threads are determined by a set of rules rather than a formula, but the effect is the same [24]. For example, threads that seem to be starved get a double quantum at the highest possible priority, and threads waiting for I/O or user input also get a priority boost.

In all cases, processes that do not use the CPU very much — such as I/O-bound processes — enjoy a higher priority for those short bursts in which they want it. This was sufficient for the interactive applications of twenty years ago. It is no longer sufficient for modern multimedia applications (a class of applications that did not exist when these schedulers were designed), because their CPU usage is relatively high.

1.2 The Resolution of Clock Interrupts

Computer systems have two clocks: a hardware clock that governs the instruction cycle, and an operating system clock that governs system activity. Unlike the hardware clock, the frequency of the system clock is not predefined: rather, it is set by the operating system on startup. Thus the system can decide for itself what frequency it wants to use. It is this tunability that is the focus of the present paper.

The importance of the system clock (also called the timer interrupt rate) lies in the fact that commodity systems measure time using this clock, including CPU usage and when timers should go off. The reason that timers are aligned with clock ticks is to simplify their implementation and bound the overhead. The alternative of setting a special interrupt for each timer event requires more bookkeeping and risks high overhead if many timers are set with very short intervals.

The most common frequency used today is 100 Hz: it is used in Linux, the BSD family, Solaris, the Windows family, and Mac OS X. This hasn't changed much in the last 30 years. For example, back in 1976 Unix version 6 running on a PDP11 used a clock interrupt rate of 60 Hz [16]. Since that time the hardware clock rate has increased by about 3 orders of magnitude, from several megahertz to over 3 gigahertz [23]. As a consequence, the size of an *operating system tick* has increased a lot, and is now on the order of 10 million cycles or instructions. Simple interactive applications such as text editors don't require that many cycles per quantum[1], making the tick rate obsolete — it is too coarse for measuring the running time of an interactive process. For example, the operating system cannot distinguish between processes that run for a thousand cycles and those that run for a million cycles, because using 100 Hz ticks on a 1 GHz processor both look like 0 time.

A special case of time measurement is setting the time

[1]Interestingly, this same consideration has also motivated the approach of making the hardware clock *slower*, rather than making the operating system clock faster as we propose. This has the benefit of reducing power consumption [10].

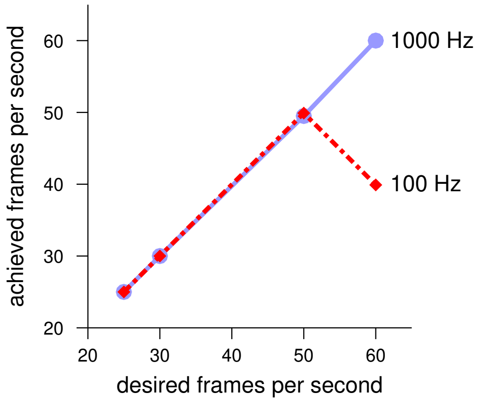

Figure 1: *Desired and achieved frame rate for the Xine MPEG viewer, on systems with 100 Hz and 1000 Hz clock interrupt rates.*

that a process may run before it is preempted. This duration, called the allocation quantum, is also measured in clock ticks. Changing the clock resolution therefore implicitly effects the quantum size. However in reality these two parameters need not be correlated, and they can be set independently of each other. The question is how to set each one.

A related issue is providing support for soft real-time applications such as games with realistic video rendering, that require accurate timing down to several milliseconds. These applications require significant CPU resources, but in a fragmented manner, and are barely served by a 100 Hz tick rate. In some cases, the limited clock interrupt rate may actually prevent the operating system from providing required services.

An example is given in Figure 1. This shows the desired and achieved frame rates of the Xine MPEG viewer showing 500 frames of a short clip that is already loaded into memory, when running on a Linux system with clock interrupt rates of 100 Hz and 1000 Hz. For this benchmark the disk and CPU power are not bottlenecks, and the desired frame rates can all be achieved. However, when using a 100 Hz system, the viewer repeatedly discards frames because the system does not wake it up in time to display them if the desired frame rate is 60 frames per second. This is an important deficiency, as 60 frames/sec is mandated by the MPEG standard.

Even finer timing services are required in other, non-desktop applications. Video rates of up to 1000 frames per second are used for recording high-speed events, such as vehicle crash experiments [26]. Similar high rates can also be expected for sampling sensors in various situations. Even higher rates are necessary in networking, for the implementation of rate-based transmission [25, 2]. Full utilization of a 100 Mb/s Fast Ethernet with 1500-byte packets requires a packet to be transmitted every 120 μs, i.e. 8333 times a second. On a gigabit link, the interval drops to 12 μs, and the rate jumps up to 83,333 times a second.

Increasing the clock interrupt rate may be expected to provide much better timing support than that available today with 100 Hz. However, this comes at the possible expense of additional overhead, and has therefore been discouraged by Linux developers (this will probably change as the 2.5 development kernel has switched to 1000 Hz for the prevalent Intel architecture; in the past, such a rate was recommended only for the Alpha processor, which according to the kernel mailing list was "strong enough to handle it") and by Sun documentation ("exercise great care if you are considering setting high-resolution ticks[2] ... this setting should never, ever be made on a production system without extensive testing first" [17, p. 56]). Our goal is to investigate this tradeoff more thoroughly.

1.3 Related Work

Other approaches to improving the soft real-time service provided by commodity systems include RT-Linux, one-shot timers, soft timers, firm timers, and priority adjustments. The RT-Linux project uses virtual machine technology to run a real-time executive under Linux, only allowing Linux to run when there are no urgent real-time tasks that need the processor [3]. Thus Linux does not run on the native hardware, but on a virtual machine. The result is a juxtaposition of a hard real-time system and a Linux system. In particular, the real-time services are not available for the Linux processes, so real time applications must be partitioned into two independent parts. However, communication between the two parts is supported.

One-shot timers do not have a pre-defined periodicity. Instead, they are set according to need. The system stores timer requests sorted by time. Whenever a timer event is fired, the system sets a timer interrupt for the next event. Variants of one-shot timers have been used in several systems, including the Pebble operating system, the Nemesis operating system for multimedia [15], and the KURT real-time system [25]. The problem is that this may lead to high overhead if many timing events are requested with fine resolution.

In soft timers the timing of system events is also not tied to periodic clock interrupts [2]. Instead, the system opportunistically makes use of convenient circumstances in order to provide higher-resolution services. For example, on each return from a system call the system may check whether any timer has become ready, and fire the respective events. As such opportunities occur at a much higher rate than the timer interrupts, the average resolution is much improved (in other words, soft timers are such a good idea specifically because the resolution of clock interrupts is so outdated). However, the timing of a specific event cannot be guaranteed, and the original low-resolution timer interrupts serve as a fallback. Using a higher clock rate, as we suggest, can guarantee a much smaller maximal deviation from the desired time.

Firm timers combine soft timers with one-shot timers [13]. This combination reduces the need for timer interrupts, alleviating the risk of excessive overheads. Firm timers together with a preemptible kernel and suitable scheduling have been shown to be effective in supporting time-sensitive applications on a commodity operating system.

Priority adjustments allow a measure of control over when processes will run, enabling the emulation of real-time services [1]. This is essentially similar to the implementation of hard real-time support in the kernel, except for the fact that it is done by an external process, and can only use the primitives provided by the underlying commodity system.

[2]This specifically means 1000 Hz.

Finally, there are also various programming projects to improve the responsiveness and performance of the Linux kernel. One is the preemptible kernel patch, which has been adopted as part of the 2.5 development kernel. It reduces interrupt processing latency by allowing long kernel operations to be preempted.

A major difference between the above approaches and ours is that they either require special APIs, make non-trivial modifications to the system, or both. Such modifications cannot be made by any user, and require a substantial review process before they are incorporated in standard software releases (if at all). For example, one-shot timers and soft timers have been known since the mid '90s, but are yet to be incorporated in a major system. By contradistinction, we focus on a single simple tuning knob — the clock interrupt rate, and investigate the benefits and the costs of turning it to much higher values than commonly done. Previous work on multimedia scheduling, with the exception of [19], has made no mention of the underlying system clock, and focused on designs for meeting deadline and latency constraints.

1.4 Preview of Results

Our goal is to show that increasing the clock interrupt rate is both possible and desirable. Measurements of the overheads involved in interrupt handling and context switching indicate that current CPUs can tolerate much higher clock interrupt rates than those common today (Section 3). We then go on to demonstrate the following:

- Using a higher tick rate allows the system to perform much more accurate billing, thus giving a better discrimination among processes with different CPU usage levels (Section 4).
- Using a higher tick rate also allows the system to provide a certain "best effort" style of real-time processing, in which applications can obtain high-resolution timing measurements and alarms (as exemplified in Figure 1, and expanded in Section 5). For applications that use time scales that are related to human perception, a modest increase in tick rate to 1000 Hz may suffice. Applications that operate at smaller time scales, e.g. to monitor certain sensors, may require much higher rates and shortening of scheduling quantum lengths (Section 7).

We conclude that improved clock resolution — and the shorter quanta that it makes possible — should be a part of any solution to the problem of scheduling soft real-time applications, and should be taken into account explicitly.

2. METHODOLOGY AND APPLICATIONS

Before presenting detailed measurement results, we first describe the experimental platform and introduce the applications used in the measurements.

2.1 The Test Platform

Most measurements were done on a 664 MHz Pentium-III machine, equipped with 256 MB RAM, and a 3DFX Voodoo3 graphics accelerator with 16 MB RAM that supports OpenGL in hardware. In addition, we performed cross-platform comparisons using machines ranging from Pentium 90 to Pentium-IV 2.4 GHz. The operating system is a 2.4.8 Linux kernel (RedHat 7.0), with the XFree86 4.1 X server. The same kernel was compiled for all the different architectures, which may result in minor differences in the generated code due to architecture-specific ifdefs. The default clock interrupt rate is 100 Hz. We modified the kernel to run at up to 20,000 Hz. The modifications were essentially straightforward, and involved extending kernel ifdefs to this range and correcting the calculation of bogomips[3].

The measurements were conducted using *klogger*, a kernel logger we developed that supports fine-grain events. While the code is integrated into the kernel, its activation at runtime is controlled by applying a special sysctl call using the /proc file system. In order to reduce interference and overhead, logged events are stored in a sizeable buffer in memory (we typically use 4 MB), and only exported at large intervals. This export is performed by a daemon that wakes up every few seconds (the interval is reduced for higher clock rates to ensure that events are not lost). The implementation is based on inlined code to access the CPU's cycle counter and store the logged data. Each event has a 20-byte header including a serial number and timestamp with cycle resolution, followed by event-specific data. The overhead of each event is only a few hundred cycles (we estimate that at 100 Hz the overhead for logging is 0.63%, at 1000 Hz it is 0.95%, and at 20,000 Hz 1.18%). In our use, we log all scheduling-related events: context switching, recalculation of priorities, forks, execs, and changing the state of processes.

2.2 The Workload

The system's behavior was measured with different clock rates and different workloads. The workloads were composed of the following applications:

- A classic interactive application — the Emacs text editor. During the test the editor was used for standard typing at a rate of about 8 characters per seconds.
- The Xine MPEG viewer, which was used to show a short video clip in a loop. Xine's implementation is multithreaded, making it a suitable representative of this growing class of applications [11]. Specifically, Xine uses 6 distinct processes. The two most important ones are the *decoder*, which reads the data stream from the disk and generates frames for display, and the *displayer*, which displays the frames at the appropriate rate. The displayer keeps track of time using alarms with a resolution of 4 ms. On each alarm it checks whether the next frame should be displayed, and if so, sends the frame to the X server. If it is too late, the frame is discarded. If it is very late, the displayer can also notify the decoder to skip certain frames.

  In the experiments, audio output was sent to /dev/null rather than to the sound card, to allow focus on interactions with the X server.
- Quake 3, which represents a modern interactive application (role playing game). Quake uses the X server's Direct Rendering Infrastructure (DRI) [21] feature which enables the OpenGL graphics library to access the hardware directly, without proxying all the requests through the X server. This results in some of the graphics processing being done by the Graphical Processor Unit (GPU) on the accelerator.

  Another interesting feature of Quake is that it is adaptive: it can change its frame rate based on how much CPU time it gets. Thus when Quake competes with other processes, its frame rate will drop. In our experiments, when running alone it is always ready to run and can use all available CPU time.
- CPU-bound processes that serve as a background load that can absorb any number of available CPU cycles, and compete with the interactive and real-time processes.

In addition, the system ran a host of default processes, mostly various daemons. Of these, the most important with regard to interactive processes is obviously the X server.

3. CLOCK RESOLUTION AND OVERHEADS

A major concern regarding increasing the clock interrupt rate is the resulting increase in overheads: with more clock interrupts more time will be wasted on processing them, and there may also be more context switches (as will be explained below in Section 6), which in turn lead to reduced cache and TLB efficiency. This is the reason why today only the Alpha version of Linux employs a rate of 1024 Hz by default. This is compounded by the concern that operating systems in general become less efficient on machines with higher hardware clock rates [20]. We will show that these concerns are unfounded, and a clock interrupt rate of 1000 Hz or more is perfectly possible.

The overhead caused by clock interrupts may be divided into two parts: direct overhead for running the interrupt handling routine, and indirect overhead due to reduced cache and TLB efficiency. The direct overhead can easily be measured using klogger. We have performed such measurements on a range of Pentium processors with clock rates from 90 MHz to 2.4 GHz, and on an Athlon XP1700+ at 1.467 GHz with DDR-SDRAM memory.

The results are shown in Table 1. We find that the overhead for interrupt processing is dropping at a much slower rate than expected according to the CPU clock rate — in fact, it is relatively stable in terms of absolute time. This is due to an optimization in the Linux implementation of gettimeofday(), whereby overhead is reduced by accessing the 8253 timer chip on each clock interrupt — rather than when gettimeofday() itself is called — and extrapolating using the cycle counter register. This takes a constant amount of time and therefore adds overhead to the interrupt handling that is not related to the CPU clock rate. Even so, the overhead is still short enough to allow many more interrupts than are used today, up to an order of 10,000 Hz. Alternatively, by removing this optimization, the overhead of clock interrupt processing can be reduced considerably, to allow much higher rates. A good compromize might be to increase the clock interrupt rate but leave the rate at which the 8253 is accessed at 100 Hz. This will amortize the overhead of the off-chip access, thus reducing the overhead per clock interrupt.

A related issue is the overhead for running the scheduler. More clock interrupts imply more calls to the scheduler.

[3]Bogomips are an estimate of the clock rate computed by the Linux kernel upon booting. The correction prevents division by zero in this calculation.

Processor	Default		Without 8253	
	Cycles	μs	Cycles	μs
P-90	814±180	9.02	498±466	5.53
PP-200	1654±553	8.31	462±762	2.32
PII-350	2342±303	6.71	306±311	0.88
PIII-664	3972±462	5.98	327±487	0.49
PIII-1.133	6377±602	5.64	426±914	0.38
PIV-2.4	14603±436	6.11	445±550	0.19
A1.467	10494±396	7.15	202±461	0.14

Table 1: *Interrupt processing overheads on different processor generations (average±standard deviation).*

Processor	Context switch		Cache BW	Trap	
	Cycles	μs	MB/s	Cycles	μs
P-90	1871±656	20.75	28±1	153±24	1.70
PP-200	1530±389	7.69	705±26	379±75	1.91
PII-350	1327±331	3.80	1314±29	343±68	0.98
PIII-664	1317±424	1.98	2512±32	348±163	0.52
PIII-1.133	1330±441	1.18	4286±82	364±278	0.32
PIV-2.4	3792±857	1.59	3016±47	1712±32	0.72
A1.467	1436±477	0.98	3962±63	274±20	0.19

Table 2: *Other overheads on different processor generations (average±standard deviation).*

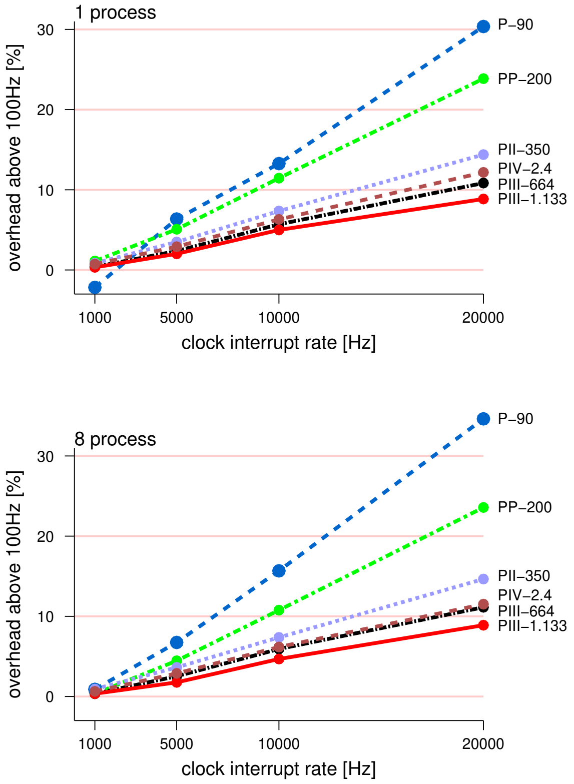

Figure 2: *Increase in overhead due to increasing the clock interrupt rate from a base case of 100 Hz. The basic quantum is 50 ms.*

More serious is the fact that in Linux the scheduler overhead is proportional to the number of processes in the ready queue. However, this only becomes an important factor for very large numbers of processes. It is also partly offset by the fact that with more ready processes it takes longer to complete a scheduling epoch, and therefore priority recalculations are done less frequently.

As a side note, it is interesting to compare clock interrupt processing overhead to other types of overhead. Ousterhout has claimed that in general operating systems do not become faster as fast as hardware [20]. We have repeated some of his measurements on the platforms listed above. The results (Table 2) show that the overhead for context switching (measured using two processes that exchange a byte via a pipe) takes roughly the same number of cycles, regardless of CPU clock speed (except on the P-IV, which is using DDR-SDRAM memory at 266 MHz and not the newer RDRAM). It therefore does become faster as fast as the hardware. We also found that the trap overhead (measured by the repeated invocation of `getpid`) and cache bandwidth (measured using `memcpy`) behave similarly. This is more optimistic than Ousterhout's results. The difference may be due to the fact that Ousterhout compared RISC vs. CISC architectures, and there is also a difference in methodology: we measure time and cycles directly, whereas Ousterhout based his results on performance relative to a MicrovaxII and on estimated MIPS ratings.

The indirect overhead of clock interrupt processing can only be assessed by measuring the *total* overhead in the context of a specific application (as was done, for example, in [2]). The application we used is sorting of a large array that occupies about half of the L2 cache (the L2 cache was 256 KB on all platforms except for the P-II 350 which had an L2 cache of 512 KB). The sorting algorithm was introsort, which is used by STL that ships with `gcc`. The sorting was done repeatedly, where each iteration first initializes the array randomly and then sorts it (but the same random sequences were used to compare the different platforms). By measuring the time per iteration under different conditions, we can factor out the added total overhead due to additional clock interrupts (as is shown below). To also check the overhead caused by additional context switching among processes, we used different multiprogramming levels, running 1, 2, 4, or 8 copies of the test application at the same time. All this was repeated for different CPU generations with different (hardware) clock rates.

Assuming that the amount of work to sort the array once is essentially fixed, measuring this time as a function of the clock interrupt rate will show how much time was added due to overhead. Figure 2 shows this added overhead as a percentage of the total time required at 100 Hz. From this we see that the added overhead at 1000 Hz is negligible, and even at 5000 Hz it is quite low. Note, however, that this is after removing the `gettimeofday()` optimization, i.e. without accessing the 8253 chip on each interrupt. For higher clock rates, the overhead increases linearly, with a slope that becomes flatter with each new processor generation (except for the P-IV). Essentially the same results are obtained with a multiporgramming level of 8. Thus we can expect higher clock interrupt rates to be increasingly acceptable.

The overhead also depends on the length of the quanta, i.e. on how much time is allocated to a process each time it runs. In Linux, the default allocation is 50 ms, which trans-

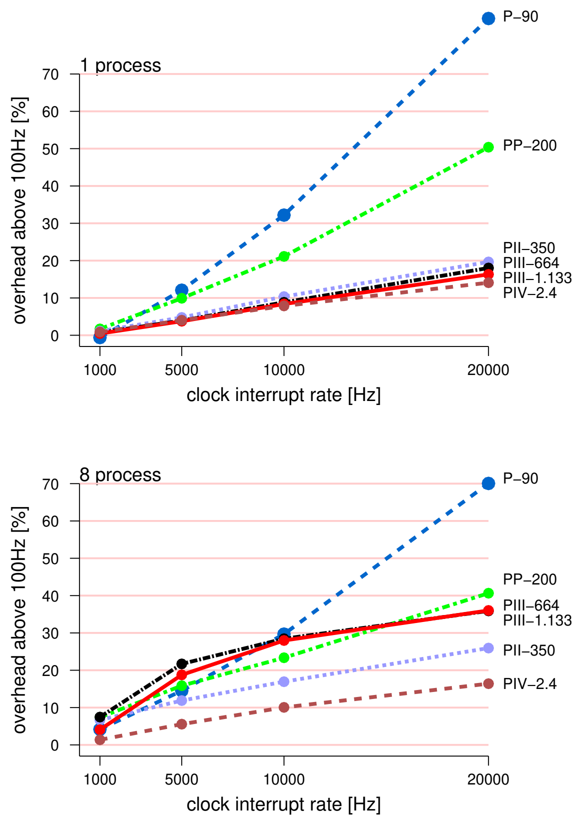

Figure 3: *Increase in overhead due to increasing the clock interrupt rate from a base case of 100 Hz. Quanta are 6 clock ticks, so they become shorter for high clock rates.*

lates to 5 ticks[4]. When raising the clock interrupt rate, the question is whether to stick with the allocation of 50 ms, or to reduce it by defining the allocation in terms of ticks, so as to improve responsiveness. The results shown in Figure 2 were for 50 ms. Figure 3 shows the same experiments when using 5 ticks, meaning that the quanta are 10 or 100 times shorter when using 1000 Hz or 10,000 Hz interrupt rates, respectively. As shown in the graphs this leads to much higher overheads, especially under higher loads, probably because there are many more context switches. This may limit the realistic clock interrupt rate to 1000 Hz or a bit more, but probably not as high as 5000 Hz (in this case the P-IV is substantially better than the other platforms, but this is due to using performance relative to 100 Hz, which was worse than for other platforms for an unknown reason). Note, however, that 1000 Hz is an order of magnitude above what is common today, and already leads to significant benefits, as shown in subsequent sections; the added overhead in this case is just a few percentage points, much less than the 10–30% which were the norm a mere decade ago [7].

Our measurements also allow for an assessment of the relative costs of direct and indirect overhead. For example, when switching from 100 Hz to 10,000 Hz, the extra time

[4]The actual allocation is 5 ticks plus one, to ensure that the allocation is strictly positive, as the 5 is derived from the integral quotient of two constants.

Application	Billing ratio		Missed quanta	
	@100Hz	@1000Hz	@100Hz	@1000Hz
Emacs	1.0746	0.9468	95.96%	73.42%
Xine	1.2750	1.0249	89.46%	74.81%
Quake	1.0310	1.0337	54.17%	23.23%
X Server°	0.0202	0.9319	99.43%	64.05%
CPU-bound	1.0071	1.0043	7.86%	7.83%
CPU+Quake	1.0333	1.0390	26.71%	2.36%

° When running Xine

Table 3: *Scheduler billing success rate.*

can be attributed to 9900 additional clock interrupts each second. By subtracting the cost of 9900 calls to the interrupt processing routine (from Table 1), we can find how much of this extra time should be attributed to indirect overhead, that is mainly to cache effects.

For example, consider the case of a P-III 664 MHz machine running a single sorting process with 50 ms quanta. The average time to sort an array once is 12.675 ms on the 100 Hz system, and 13.397 ms on the 10,000 Hz system. During this time the 10,000 Hz system suffered an additional $9900 \times 0.013397 = 133$ interrupts. According to Table 1 the overhead for each one (without accessing the 8253 chip) is 0.49 μs, so the total additional overhead was $133 \times 0.49 = 65\mu$s. But the difference in the time to sort an array is $13397 - 12675 = 722\mu$s! Thus $722 - 65 = 657\mu$s are unaccounted for, and should be attributed to cache effects and scheduler overhead. In other words, $657/722 = 91\%$ of the overhead is indirect, and only 9% is direct. This number is typical of many of the configurations checked. The indirect overhead on the P-IV and Athlon machines, and when using shorter quanta on all machines, are higher, and may even reach 99%. This means that the figures given in Table 1 should be multiplied by at least 10 (and in some extreme cases by as much as 100) to derive the real cost of increasing the clock interrupt rate.

4. CLOCK RESOLUTION AND BILLING

Practically all commodity operating systems use priority-based schedulers, and factor CPU usage into their priority calculations. CPU usage is measured in ticks, and is based on sampling: the process running when a clock interrupt occurs is billed for this tick. But the coarse granularity of ticks implies that billing may be inaccurate, leading to inaccurate information used by the scheduler.

The relationship between actual CPU consumption and billing on a 100 Hz system is shown at the top of Figure 4. The X axis in these graphs is the effective quantum length: the exact time from when the process is scheduled to run until when it is preempted or blocked. While the effective quantum tends to be widely distributed, billing is done in an integral numbers of ticks. In particular, for Emacs and X the typical quantum is very short, and they are practically never billed!

Using klogger, we can tabulate all the times each application is scheduled, for how much time, and whether or not this was billed. The data is summarized in Table 3. The billing ratio is the time for which an application was billed by the scheduler, divided by the total time actually consumed by it during the test. The miss percentage is the percentage

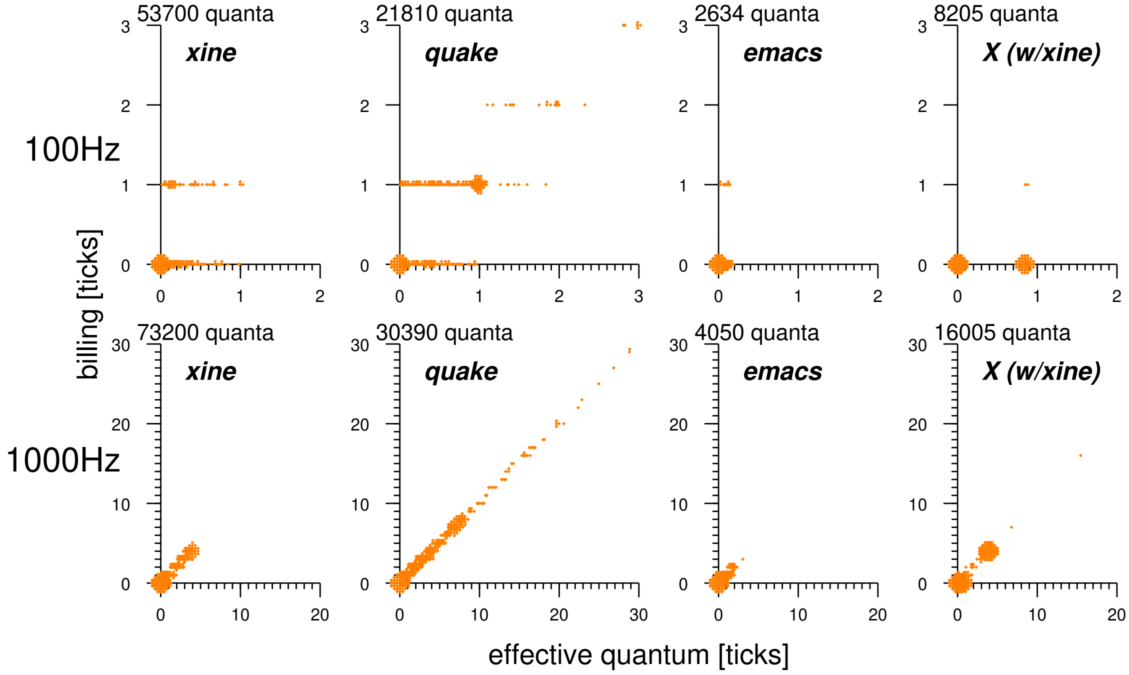

Figure 4: *The relationship between effective quanta durations and how much the process is billed, for different applications, using a kernel running at 100 Hz and at 1000 Hz. Concentrations of data points are rendered as larger disks; otherwise the graphs would have a clean steps shape, because the billing (Y axis) is in whole ticks. Note also that the optimal would be a diagonal line with slope 1.*

of the application's quanta that were totally missed by the scheduler and not billed for at all.

The table shows that even though very many quanta are totally missed by the scheduler, especially for interactive applications, most applications are actually billed with reasonable accuracy in the long run. This is a result of the probabilistic nature of the sampling. Since most of the quanta are shorter than one clock tick, and the scheduler can only count in complete tick units, many of the quanta are not billed at all. But when a short quantum does happen to include a clock interrupt, it is over billed and charged a full tick. On average, these two effects tend to cancel out, because the probability that a quantum includes a tick is proportional to its duration. The same averaging happens also for quanta that are longer than a tick: some are rounded up to the next whole tick, while others are rounded down.

A notable exception is the X server when running with Xine (we used Xine because it intensively uses the X server, as opposed to Quake which uses DRI). As shown below in Section 6, when running at 100 Hz this application has quanta that are either extremely short (around 68% of the quanta), or 0.8–0.9 of a tick (the remaining 32%). Given the distribution of quanta, we should expect over 30% of them to include a tick and be counted. But the scheduler misses over 99% of them, and only bills about 2% of the consumed time! This turns out to be the result of synchronization with the operating system ticks. Specifically, the long quanta always occur after a very short quantum of a Xine process that was activated by a timer alarm. This is the displayer, which checks whether to display the next frame. When it decides that the time is right, it passes the frame to X. The X server then awakes and takes a relatively long time to actually display the frame, but just less than a full tick. As the timer alarm is carried out on a tick, these long quanta always start very soon after one tick, and complete just before the next tick. Thus, despite being nearly a tick long, they are hardly ever counted.

When running the kernel at 1000 Hz we can see that the situation improves dramatically — the effective quantum length, even for interactive applications, is typically several ticks long, so the scheduler bills the process an amount that reflects the actual consumed time much more accurately. In particular, on a 1000 Hz system X is billed for over 93% of the time it consumed, with the missed quanta percentage dropping to 64% — the fraction of quanta that are indeed very short.

An alternative to this whole discussion is of course the option to measure runtime accurately, rather than sampling on clock interrupts. This can be done easily by accessing the CPU cycle counter [6]. However, this involves modifying the operating system, whereas we are only interested in the effects obtainable by simple tuning of the clock interrupt rate.

5. CLOCK RESOLUTION AND TIMING

Increasing the kernel's clock resolution also yields a major benefit in terms of the system's ability to provide accurate timing services. Specifically, with a high-resolution clock it

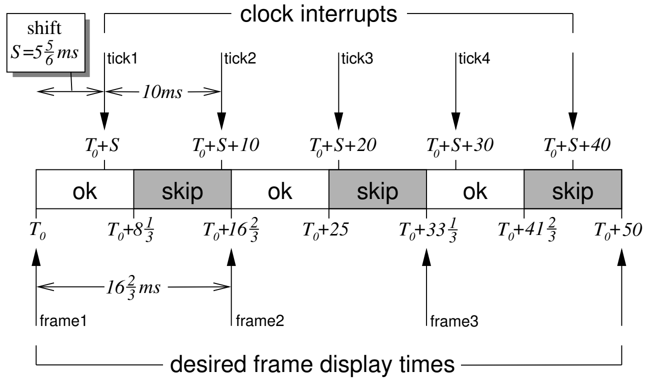

Figure 5: *Relationship of clock interrupts to frame display times that causes frames to be skipped. In this example the relative shift is $5\frac{5}{6}$ ms, and frame 2 is skipped.*

Application	Quanta/sec @100Hz	@1000Hz
Emacs	22.36	34.60
Xine (all processes)	470.67	695.94
Quake	187.88	273.85
X Server (w/Xine)	71.35	148.21
CPU-bound	28.81	38.97

Table 4: *Average quanta per second achieved by each application when running in isolation.*

Application	CPU usage @100Hz	@1000Hz
Xine	39.42%	40.42%
X Server	20.10%	20.79%
idle loop	31.46%	31.58%
other	9.02%	7.21%

Table 5: *CPU usage distribution when running Xine.*

is possible to deliver high-resolution timer interrupts. This is especially significant for soft real-time applications such as multimedia players, which rely on timer events to keep correct time.

A striking example was given in the introduction, where it was shown that the Xine MPEG player was sometimes unable to display a movie at a rate of 60 frames per second (which is mandated by the MPEG standard). This is somewhat surprising, because the underlying system clock rate is 100 Hz — higher than the desired rate.

The problem stems from the relative timing of the clock interrupts and the times at which frames are to be displayed. Xine operates according to two rules: it does not display a frame ahead of its time, and it skips frames that are late by more than half a frame duration. A frame will therefore be displayed only if the clock interrupt that causes Xine's timer signal to be delivered occurs in the first half of a frame's scheduled display time. In the case of 60 frames per second on a 100 Hz system, the smallest common multiple of the frame duration ($\frac{1000}{60} = 16\frac{2}{3}$ ms) and clock interval (10 ms) is 50 ms. Such an interval is shown in Figure 5. In this example frame 2 will be skipped, because interrupt 2 is a bit too early, whereas interrupt 3 is already too late. In general, the question of whether this will indeed happen depends on the relative shift between the scheduled frame times and the clock interrupts. A simple inspection of the figure indicates that frame 1 will be skipped if the shift (between the first clock interrupt and the first frame) is in the range of $8\frac{1}{3}$–10 ms, frame 2 will be skipped for shifts in the range 5–$6\frac{2}{3}$ ms, and frame 3 will be skipped for shifts in the range $1\frac{2}{3}$–$3\frac{1}{3}$ ms, for a total of 5 ms out of the 10 ms between ticks. Assuming the initial shift is random, there is therefore a 50% chance of entering a pattern in which a third of the frames are skipped, leading to the observed frame rate of about 40 frames per second (in reality, though, this happens much less than 50% of the time, because the initial program startup tends to be synchronized with a clock tick).

To check this analysis we also tried a much more extreme case: running a movie at 50 frames per second on a 50 Hz system. In this case, either all clock interrupts fall in the first half of their respective frames, and all frames are shown, or else all interrupts fall in the second half of their frames, and all are skipped. And indeed, we observed runs in which all frames were skipped and the screen remained black throughout the entire movie.

The implication of the above is that the timing service has to have much finer resolution than that of the requests. For Xine to display a movie at 60 Hz, the timing service needs a resolution of 4 ms. This is required for the application to function correctly, not for the actual viewing, and therefore applies despite the fact that this clock resolution is much higher than the screen refresh rate.

6. CLOCK RESOLUTION AND THE INTERLEAVING OF APPLICATIONS

Recall that we define the effective quantum length to be the interval from when a process is scheduled until it is descheduled for some reason. On our Linux system, the allocation for a quantum is 50 ms plus one tick. However, as we can see from Figures 4 and 6 (introduced below), our interactive applications never even approach this limit. They are always preempted or blocked much sooner, often quite soon in their first tick. In other words, the effective quantum length is very short. This enables the system to support more than 100 quanta per second, even if the clock interrupt rate is only 100 Hz, as shown in Table 4. It also explains the success of soft timers [2].

The distributions of the effective quantum length for the different applications are shown in Figure 6, for 100 Hz and 1000 Hz systems. An interesting observation is that when running the kernel at 1000 Hz the effective quanta become even shorter. This happens because the system has more opportunities to intervene and preempt a process, either because it woke up another process that has higher priority, or due to a timer alarm that has expired. However, the total CPU usage does not change significantly (Table 5). Thus increasing the clock rate did not change the amount of computation performed, but the way in which it is partitioned into quanta, and the granularity at which the processes are interleaved with each other.

A specific example is provided by Xine. One of the Xine processes sets a 4 ms alarm, that is used to synchronize the video stream. In a 100 Hz system, the alarm signal is only delivered every 10 ms, because this is the size of a tick. But when using a 1000 Hz clock the system can actually

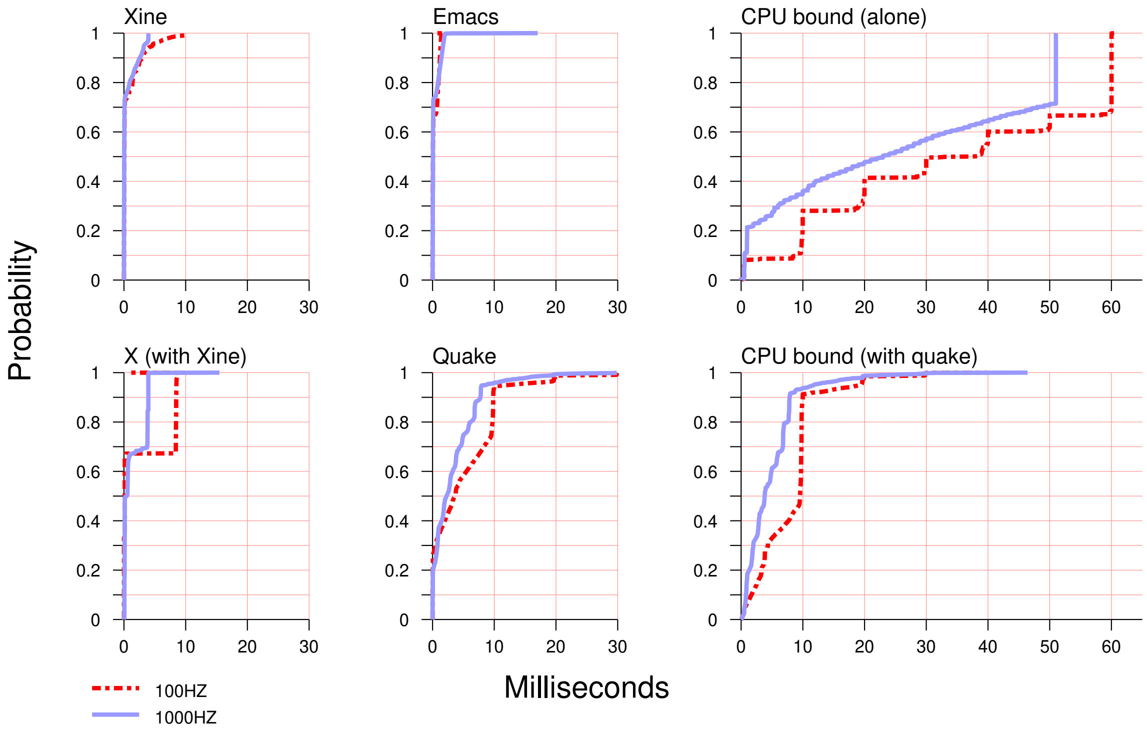

Figure 6: *Cumulative distribution plots of the effective quantum durations of the different applications.*

deliver the signals on time. As a result the maximal effective quanta of X and the other Xine processes are reduced to 4 ms, because they get interrupted by the Xine process with the 4 ms timer.

Likewise, the service received by CPU-bound applications is not independent of the interactive processes that accompany them. To investigate this effect, these processes were measured alone and running with Quake. When running alone, their quanta are typically indeed an integral number of ticks long. Most of the time the number of ticks is less than the full allocation, due to interruptions from system daemons or klogger, but a sizeable fraction do achieve the allocated 50 ms plus one tick (which is an additional 10 ms at 100 Hz, but only 1 ms at 1000 Hz). But when Quake is added, the quanta of the CPU-bound processes are shortened to the same range as those of Quake, and moreover, they become less predictable. This also leads to an increase in the number of quanta that are missed for billing (Table 3), unless the higher clock rate of 1000 Hz is used.

7. TOWARDS BEST-EFFORT SUPPORT FOR REAL-TIME

In this section we set out to explore how close a general purpose system can come to supporting real-time processes in terms of timing delays, only by tuning the clock interrupt rate and reducing the allocated quanta. The metric that we use in order to perform such an evaluation is latency: the difference between the time in which an alarm requested by a process *should* expire, and the time in which this process was actually assigned a CPU.

Without worrying about overhead (for the moment), our aim is to show that under loads of up to 8 processes, we can bound the latency to be less than 1 millisecond. As there are very many types of soft real-time applications, we sample the possible space by considering three types of processes:

1. *BLK*: A process repeatedly sets alarms without performing any type of computation. Our experiments involved processes that requested an alarm signal 500 times, with delays that are uniformly distributed between 1 and 1000 milliseconds.

2. *N%*: Same as *BLK*, with the difference that a process computed for a certain fraction (*N%*) of the time till the next alarm. Specifically, we checked computation of $N = 1$, 2, 4, and 8% out of this interval. Note for example that a combination of 8 processes computing for 8% of the time leads to an average of 64% CPU utilization. To check what happens when the CPU is not left idle, we also added CPU-bound processes that do not set timers.

3. *CONT*: Same as *N%* where N=100% i.e. the process computes continuously.

For each of the above 3 types, we checked combinations of 1, 2, 4, and 8 processes. All the processes that set timers were assigned to the (POSIX) Round-Robin class. Note that a combination of more than one *CONT*-process constitutes the worst-case scenario, because — contrary to the other workloads — the CPU is always busy and there are always alternative processes with similar priorities (in the Round-Robin queue) that are waiting to run.

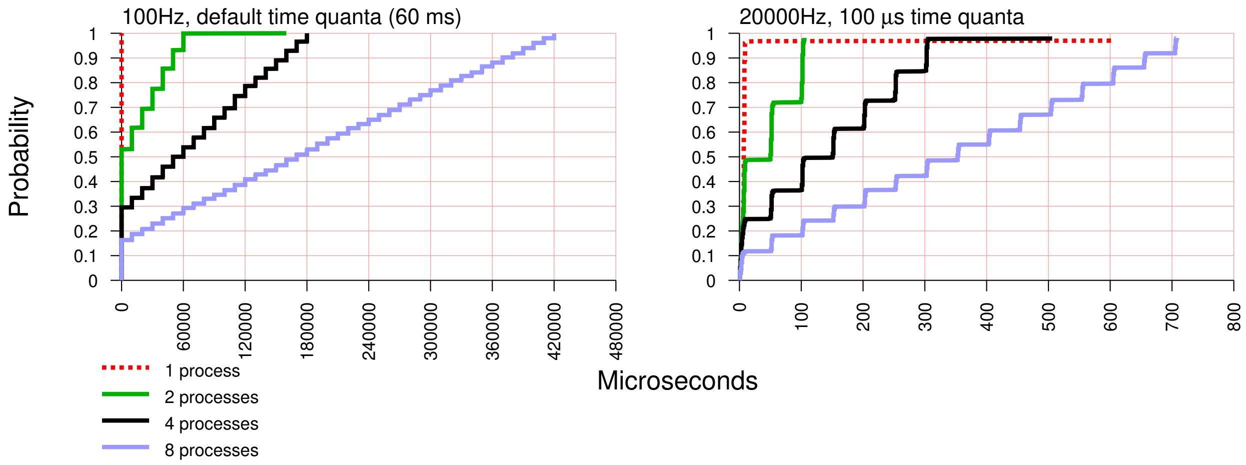

Figure 7: *Distributions of latencies till a timer signal is delivered, for processes that compute continuously and also set timers for random intervals of up to one second.*

The base system we used is the default configuration of Linux, with 100 Hz clock interrupt rate and a 60 ms (6 ticks) maximal quantum duration. In order to achieve our sub-millisecond latency goal, we compared this with a rather aggressive alternative: 20,000 Hz clock interrupt rate and 100 μs (2 ticks) quantum (note that we are changing *two* parameters at once: both the clock resolution and the number of ticks in a quantum). Theoretically, for this configuration the maximal latency would be $100\mu s \times 7 = 700\mu s < 1$ ms, because even if a process is positioned at the end of the run-queue it only needs to wait for seven other processes to run for $100\mu s$ each.

The results shown in Figure 7 confirm our expectations. This figure is associated with the worst-case scenario of a workload composed solely of *CONT* processes. Examining the results for the original 100 Hz system (left of Figure 7), we see that a single process receives the signal within one tick, as may be expected. When more processes are present, there is also a positive probability that a process will nevertheless receive the signal within a tick: $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$ for 2, 4 and 8 processes, respectively. The Y-axis of the figure shows that the actual fractions were 0.53, 0.30, and 0.16 (respectively), slightly more than the associated probabilities. But, a process may also be forced to wait for other processes that precede it to exhaust their quanta. This leads to the step-like shape of the graphs, because the wait is typically an integral number of ticks. The maximal wait is a full quantum for each of the other processes. In the case of 8 competing processes, for example, the maximum is 60 ms for each of the other 7, for a total of 420 ms (=420,000 μs).

The situation on the 20,000 Hz system is essentially the same, except that the time scale is much much shorter — the latency is almost always less than a millisecond, as expected. In other words, the high clock interrupt rate and rapid context switching allow the system to deliver timer signals in a timely manner, despite having to cycle through all competing processes.

Table 6 shows that this is the case for all our experiments (for brevity only selected experiments are shown). Note that using the higher clock rate also provides significantly improved latencies to the experiments where processes only compute for a fraction of the time till the timer event. With 100 Hz even this scenario sometimes causes conflicts, despite the relatively low overall CPU utilization. The relatively few long-latency events that remain in the high clock-rate case are attributed to conflicts with system daemons that perform disk I/O, such as the pager. Similar effects have been noted in other systems [14]. These problems are expected to go away in the next Linux kernel, which is preemptive; they should not be an issue in other kernels that are already preemptive (such as Solaris).

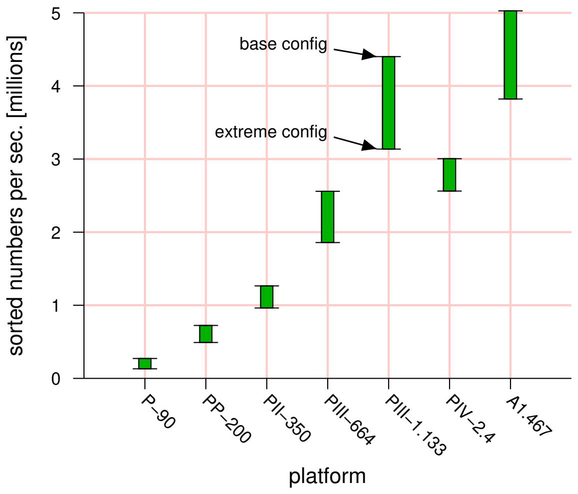

Figure 8: *Throughput of the sort application, measured as how many millions of numbers were sorted per second, with 8 competing processes.*

But what about overheads? As shown in Figure 3, when running continuously computing processes (in that case, a sorting application) with a 20,000 Hz clock interrupt rate and quanta of 6 ticks, the additional overhead can reach 35% on contemporary architectures. The overhead for the shorter 2-tick quanta used here may be even higher. This

Processes		@100Hz				@20,000Hz			
Type	Number	0.9	0.95	0.99	max	0.9	0.95	0.99	max
BLK	2	5	8	11	40	13	14	21	23
BLK	8	5	12	22	420	7	9	13	25
CONT	2	50,003	60,003	60,004	160,006	102	103	18,468	60,448
CONT	8	370,014	400,014	420,015	740,025	656	706	15,096	68,139
2%	2	6	9	9,193	19,153	13	15	23	837
2%	8	2,910	8,419	17,940	32,944	12	52	53	1,809
8%	2	9	12,431	39,512	60,003	14	19	53	3,797
8%	8	40,003	60,005	130,006	294,291	53	53	54	37,328
4%	1+2CPU	50,003	50,003	50,004	50,005	55	56	200	256
4%	1+8CPU	50,003	50,003	170,014	280,010	56	57	59	856

Table 6: *Tails of distributions of latencies to deliver timer signals in different experimental settings. Table values are latencies in microseconds, for various percentiles of the distribution.*

seems like an expensive and unacceptable price to pay. However, if we examine the application throughput on different platforms the picture is not so bleak. Figure 8 compares the achieved throughput, as measured by numbers sorted per second, for two configurations. The base configuration uses 100 Hz interrupts and 60 ms quanta. The extreme configuration uses 20,000 Hz interrupts and 100 μs quanta. While performance dramatically drops when comparing the two configurations on the same platform, the extreme configuration of each platform still typically outperforms the base configuration on the previous platform. For example, PIII-664 running the base configuration manages to sort about 2,559,000 numbers per second, while the PIII-1.133 with the extreme configuration sorts about 3,136,000 numbers per second (the P-IV consistently performs worse than previous generations). This is an optimistic result which means that in order to get the same or even improve the performance of an existing platform, while achieving sub-millisecond latency, all one has to do is upgrade to the next generation. This is usually *much* cheaper than purchasing the industrial hard real-time alternative.

8. CONCLUSIONS AND FUTURE WORK

General purpose systems, such as Linux and Windows, are already often used for soft real-time applications such a viewing video, playing music, or burning CDs. Other less common applications include various control functions, ranging from laboratory experiment control to traffic-light control. Such applications are not critical to the degree that they require a full-fledged real-time system. However, they may face problems on a typical commodity system due to the lack of adequate support for high-resolution timing services. A special case is "timeline gaps", where the processor is totally unavailable for a relatively long time [14].

Various solutions have been proposed for this problem, typically based on explicit support for timing functions. In particular, very good results are obtained by using soft timers or one-shot timers. The idea there is to change the kernel's timing mechanism from the current periodic time sampling to event-based time sampling. However, since this event-based approach calls for a massive redesign of a major kernel subsystem, it has remained more of an academic exercise and has yet to make it into the world of mainstream operating systems.

The goal of this paper is to check the degree to which existing systems can provide reasonable soft real-time services, specifically for interactive applications, just by leveraging the very fast hardware that is now routinely available, without any sophisticated modifications to the system. The mechanism is simply to increase the frequency of the periodic timer sampling. We show that this solution — although suffering from non-negligible overhead — is a viable solution on today's ultra-fast CPUs. We also show that implementing this solution in mainstream operating systems is as trivial as turning a tuning knob, possibly even at system runtime.

We started with the observation that there is a large and growing gap between the CPU clock rates, which grow exponentially, and the system clock interrupt rates, which are rather stable at 100 Hz. We showed that by increasing the clock interrupt rate by a mere order of magnitude, to 1000 Hz, one achieves significant advantages in terms of timing and billing services, while keeping the overheads acceptably low. The modifications required to the system are rather trivial: to increase the clock interrupt rate, and reduce the default quantum length. As multimedia applications typically operate in this range (i.e. with timers of several milliseconds), such an increase may be enough to satisfy this important class of applications. A similar observation has been made by Nieh and Lam with regard to the scheduling of multimedia applications in the SMART scheduler [19]. A rate of 1000 Hz is used in the experimental Linux 2.5 kernel, and also on personal systems of some kernel hackers [12].

For more demanding applications, we experimented with raising the clock interrupt rate up to 20,000 Hz, and found that by doing so applications are guaranteed to receive timer signals within one millisecond of the correct times with high probability, even under loaded conditions.

In addition to suggesting that 1000 Hz be used as the minimal default clock rate, we also propose that the HZ value and the quantum length be settable parameters, rather than compiled constants. This will enable users of systems that are dedicated to a time-sensitive task to configure them so as to bound the latency, by shortening the quantum so that when multiplied by the expected number of processes in the system the product is less than the desired bound. Of course, this functionality has to be traded off with the overhead it entails. Such detailed considerations can only be made by knowledgeable users on a case-by-case basis. Even so, this is expected to be cost effective relative to the alternative of procuring a hard real-time system.

The last missing piece is the correct prioritization of applications under heavy load conditions. The problem is that modern interactive applications may use quite a lot of CPU power to generate realistic graphics and video in real-time, and may therefore be hard to distinguish from low priority CPU-bound applications. This is especially hard when faced with multi-threaded applications (like Xine), or if applications are adaptive (as Quake is) and can always use additional compute power to improve their output. Our future work therefore deals with alternative mechanisms for the identification of interactive processes. The mechanisms we are considering involve tracking the interactions of applications with the X server, and thus with input and output devices that represent the local user [9].

Acknowledgements

Many thanks are due to Danny Braniss and Tomer Klainer for providing access to various platforms and helping make them work.

9. REFERENCES

[1] B. Adelberg, H. Garcia-Molina, and B. Kao, "*Emulating soft real-time scheduling using traditional operating system schedulers*". In *Real-Time System Symp.*, Oct 1994.

[2] M. Aron and P. Druschel, "*Soft timers: efficient microsecond software timer support for network processing*". *ACM Trans. Comput. Syst.* **18(3)**, pp. 197–228, Aug 2000.

[3] M. Barabanov and V. Yodaiken, "*Introducing real-time Linux*". *Linux Journal* **34**, Feb 1997. http://www.linuxjournal.com/article.php?sid=0232.

[4] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*. Addison-Wesley, 2nd ed., 1998.

[5] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.

[6] J. B. Chen, Y. Endo, K. Chan, D. Mazières, A. Dias, M. Seltzer, and M. D. Smith, "*The measured performance of personal computer operating systems*". *ACM Trans. Comput. Syst.* **14(1)**, pp. 3–40, Feb 1996.

[7] R. T. Dimpsey and R. K. Iyer, "*Modeling and measuring multiprogramming and system overheads on a shared memory multiprocessor: case study*". *J. Parallel & Distributed Comput.* **12(4)**, pp. 402–414, Aug 1991.

[8] K. J. Duda and D. R. Cheriton, "*Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*". In 17th *Symp. Operating Systems Principles*, pp. 261–276, Dec 1999.

[9] Y. Etsion, D. Tsafrir, and D. G. Feitelson, *Human-Centered Scheduling of Interactive and Multimedia Applications on a Loaded Desktop*. Technical Report, Hebrew University, Mar 2003.

[10] K. Flautner and T. Mudge, "*Vertigo: automatic performance-setting for Linux*". In 5th *Symp. Operating Systems Design & Implementation*, pp. 105–116, Dec 2002.

[11] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "*Thread-level parallelism and interactive performance of desktop applications*". In 9th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 129–138, Nov 2000.

[12] FreeBSD Documentation Server, *Thread on "clock granularity (kernel option HZ)"*. URL http://docs.freebsd.org/mail/archive/2002/freebsd-hackers/20020203.freebsd-hackers.html, Feb 2002.

[13] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "*Supporting time-sensitive applications on a commodity OS*". In 5th *Symp. Operating Systems Design & Implementation*, pp. 165–180, Dec 2002.

[14] J. Gwinn, "*Some measurements of timeline gaps in VAX/VMS*". *Operating Syst. Rev.* **28(2)**, pp. 92–96, Apr 1994.

[15] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "*The design and implementation of an operating system to support distributed multimedia applications*". *IEEE J. Select Areas in Commun.* **14(7)**, pp. 1280–1297, Sep 1996.

[16] J. Lions, *Lions' Commentary on UNIX 6th Edition, with Source Code*. Annabooks, 1996.

[17] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, 2001.

[18] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "*SVR4 UNIX scheduler unacceptable for multimedia applications*". In 4th *Int'l Workshop Network & Operating System Support for Digital Audio and Video*, Nov 1993.

[19] J. Nieh and M. S. Lam, "*The design, implementation and evaluation of SMART: a scheduler for multimedia applications*". In 16th *Symp. Operating Systems Principles*, pp. 184–197, Oct 1997.

[20] J. K. Ousterhout, "*Why aren't operating systems getting faster as fast as hardware?*". In *USENIX Summer Conf.*, pp. 247–256, Jun 1990.

[21] B. Paul, "*Introduction to the Direct Rendering Infrastructure*". http://dri.sourceforge.net/doc/DRIintro.html, August 2000.

[22] M. A. Rau and E. Smirni, "*Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads*". In *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 252–261, Oct 1999.

[23] R. Ronen, A. Mendelson, K. Lai, S-L. Lu, F. Pollack, and J. P. Shen, "*Coming challenges in microarchitecture and architecture*". *Proc. IEEE* **89(3)**, pp. 325–340, Mar 2001.

[24] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000*. Microsoft Press, 3rd ed., 2000.

[25] 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 4th *IEEE Real-Time Technology & App. Symp.*, pp. 112–119, Jun 1998.

[26] D. Tyrell, K. Severson, A. B. Perlman, B. Brickle, and C. Vaningen-Dunn, "*Rail passenger equipment crashworthiness testing requirements and implementation*". In *Intl. Mechanical Engineering Congress & Exposition*, Nov 2000.