General-Purpose Timing: The Failure of Periodic Timers

Dan Tsafrir   Yoav Etsion   Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

Abstract

All general-purpose commodity operating systems use periodic clock interrupts to regain control and measure the passage of time. This is ill-suited for desktop settings, as the fine-grained timing requirements of modern multimedia applications require a high clock rate, which may suffer from significant overhead. It is ill-suited for HPC environments, as asynchronous interrupts ruin the coordination among cluster nodes. And it is ill-suited for mobile platforms, as it wastes significant energy, especially when the system is otherwise idle. To be truly general-purpose, systems should therefore switch to a mechanism that is closer to one-shot timers (set only for specific needs) while avoiding the potentially huge overhead they entail. With a careful design it is possible to achieve both high accuracy and low overhead, thus significantly extending the applicability of general-purpose operating systems.

1 Introduction

In recent years it has became increasingly popular to use commodity operating systems (OSs) in contexts traditionally requiring specialized and usually expensive proprietary software. For example, nowadays Linux is deployed on a huge variety of systems ranging from as little as mobile phones, cameras, and PDAs, to as large as supercomputers [16, 2]. This situation places the focus on the term "general" when considering "general purpose" OSs, as they are required to reasonably support an unforeseen wide range of applications with different and sometimes contradicting needs. Within this context, the timing services provided by the OS pose major problems, with different domains pulling in different directions.

A major consideration when designing a timing service is accuracy, which is important in the desktop domain, with various multimedia applications that require millisecond resolution. Providing this by frequent periodic interrupts creates substantial overhead, especially in HPC environments. In this context, interrupting the application on one node may cause delays on all the other nodes in a cluster, thus amplifying the detrimental effect. Reducing unnecessary activity is also desirable for mobile platforms, where energy conservation is of prime importance.

We have found that these three domains are sometimes unaware of each other and offer conflicting solutions. We therefore ask: can the different considerations be reconciled? Somewhat surprisingly, it seems the answer is yes. The problems stem from a 30-years-old design decision: the use of periodic timers. At boot time, a general-purpose kernel (all Windows and Unix flavors) sets a hardware clock to generate periodic interrupts every few milliseconds (this constant time interval is called a *tick*). The interrupts invoke a kernel routine (called the *tick handler*) responsible for important OS activities such as accounting for the CPU time used by the current process, designating it for preemption if its quantum is exhausted, or notifying the process if it has pending signals. The practical meaning of this is that general-purpose OSs are based on *polling*. While this was a good design decision in the 1970s, things have changed, and its drawbacks are accumulating into a critical mass suggesting the price of polling is becoming too high, and that it's time to reconsider.

2 The Problem with Polling

2.1 Inaccuracy and Overhead

Until recently, 100 Hz was the common default tick frequency, used by Linux, the BSD family, Solaris, IRIX, the Windows family, Mac OS X, and more. This value hasn't changed much since the dawn of general purpose OSs. For example, in 1976, Unix 6 running on a PDP11 used a tick rate of 60 Hz [10]. The problem with 100 Hz rate is that soft realtime applications such as movie players or games with realistic video rendering require accurate timing down to milliseconds, and a 100 Hz resolution is not enough.

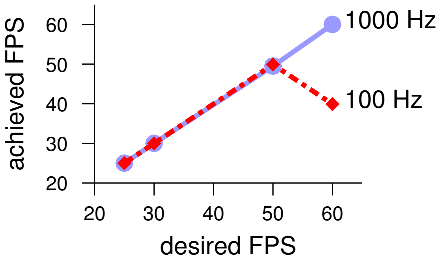

Figure 1: *Desired and achieved frames per second (FPS) for the Xine MPEG viewer, on systems with 100 Hz and 1000 Hz tick rates.*

Fig. 1 is a striking example [3]. It shows desired and achieved frame rates of the Xine MPEG viewer for 500 frames of a memory-resident clip, when running on a Linux system with 100 Hz and 1000 Hz ticks. As the memory and CPU power are not bottlenecks in this case, display rates mandated by the MPEG standard (25, 30, 50, and 60 FPS) can all in principle be achieved. However, when using a 100 Hz system to display 60 FPS, Xine might repeatedly discard frames due to a timing misalignment — similar to the requirement for a sampling-frequency greater than twice the bandwidth of the original signal in Shannon's sampling theorem.

Consequently, there is a growing trend of increasing the tick rate to 1000 Hz (Linux, FreeBSD, DragonFlyBSD). But even finer timing services are required in other, non-desktop applications. Video rates of up to 1000 FPS are used for recording high-speed events, such as vehicle crash experiments [15]. 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 [1]: 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.

A general-purpose operating system that aims to support such applications must increase its tick rate significantly. This of course comes at the price of additional overhead, both direct (context switching from the running process to the tick handler and back; executing the handler) and indirect (resulting cache and TLB pollution). A tick frequency sufficient for obtaining sub-millisecond latency under loaded condition incurs an unacceptable overhead penalty of up to half the throughput [3]. Section 3 will survey other solutions to this problem and show that these are either inadequate for general purpose OSs, or conflict with the domains described next.

2.2 Asynchrony and Overhead

Clusters are becoming very popular for high-performance computation (HPC). Such systems typically employ a general-purpose OS on each node (e.g. in last year's Top500 supercomputer list ten of the top twenty machines ran Linux). Parallel jobs are executed on such clusters by spawning one process per CPU, and running to completion with no interference. HPC applications are often bulk-synchronous which means each participating process is composed of iterative computation phases separated by barriers (where speedy processes wait for lagging ones to catch up). After each barrier, processes perform some communication before moving to the next phase. The granularity, or time it takes to complete a single computation phase, can be a millisecond or even less for real world applications [12].

In recent years, several independent studies revealed a peculiar phenomenon: fine-grained jobs running on a cluster composed of 4-way SMP nodes terminate faster if they explicitly waste resources and leave one (or more) processors idle per node [13, 12, 7]. This phenomenon was traced to a variability in the duration of computation phases: In most cases phases take a constant time to complete, but sometimes these are prolonged due to individual system activity of nodes. This activity is assigned to an idle CPU if such exits, but will otherwise preempt a running process of the parallel job. The delay is arguably unnoticeable for a sequential application. But in the context of a bulk-synchronous job, the price is dramatically amplified, as all the other processes across the entire cluster must wait for the delayed process to catch up and complete the barrier.

In an attempt to quantify the extent of this phenomenon we came up with the following simplistic model: Let n be the number of nodes used by a job. For a given node, let p be the probability that a process running on it is delayed in the current phase. Assuming independence, a job's probability to complete the current phase with no delay is $(1-p)^n$. As shown in Fig. 2, if a job is to complete most phases

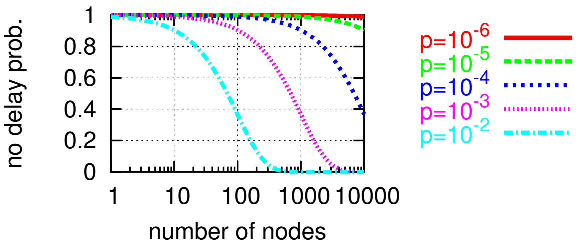

Figure 2: *The probability a job isn't delayed in the current computation phase, as a function of its size:* $(1 - p)^n$

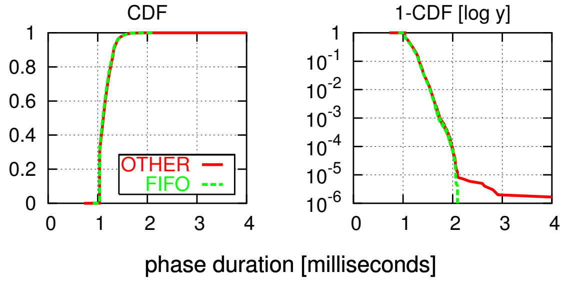

Figure 3: *Cumulative Distribution Function (CDF) of time actually taken to run a loop designed to take 1ms, under the default (OTHER) and realtime (FIFO) scheduling priorities. The right plot focuses on the tail by showing the survival function on a log scale.*

with no delays, p should be at least an order of magnitude smaller than $1/n$. For example, if a cluster has 100 nodes, we should aspire to reduce p below 10^{-3}.

In reality p depends on the job's granularity. For the purpose of this paper, let's assume this is 1 ms. We have calibrated an empty loop (a computation phase) to finish after 1 ms, and ran it a million times on a Pentium-IV 2.8GHz Linux machine with 1000 Hz ticks, saving a cycle-resolution timestamp after each phase. No other user processes were executing. At the end of the benchmark we computed the duration of each phase by subtracting successive measurements. The results – shown in Fig. 3 – are very disturbing. Most phases take 1 ms or a bit more, but some are longer than 1.6 ms with probability of $p = 10^{-2}$. According to Fig. 2, this means clusters of only tens of nodes will almost certainly suffer at least one node with such a delay in each phase, causing a global slowdown factor of 1.6. Larger supercomputers could suffer doubling of runtime.

When using the default OTHER scheduler the situation is especially bad, as system processes might interfere; With the realtime FIFO scheduling, only system interrupts are able to interfere. Instrumenting the kernel to log all interrupts revealed that the only activity present in the system while the measurements took place were about a million ticks and 3,000 network interrupts, indicating ticks are probably the main cause of the problem. This was verified by repeating the measurements with kernels compiled with 100 and 10 Hz ticks, which experienced far smaller time variability, respectively. But measuring direct overhead of the tick handler indicated that it only accounts for 0.8% of available cycles (using the data from Fig. 3, indirect overhead is found to be about 14% — significant even for a uniprocessor). We therefore concluded that most of the effect is indirect overhead, due to cache misses. This was verified by repeating the experiment with the cache disabled. In this case, subtracting interrupts' direct overhead from the duration of the phases in which they occurred resulted in a perfectly vertical CDF.

To conclude, our findings indicate that the observed variability is a product of ticks, network interrupts, and the cache effects they cause. However, many clusters employ high-end communication networks (Myrinet or Quadrics) that provide dedicated processors for handling of network events. Such hardware eliminates network interrupts as a source of variability, leaving periodic ticks as the major source of degraded performance.

2.3 Power Consumption

A major consideration for handheld and mobile devices is the conservation of energy. Handling each tick requires the expenditure of some energy — 20W for a MIPS processor according to Li and John [9], not counting additional energy spent to re-populate caches. In fact, this is not only a consideration for mobile devices: the energy wasted by general purpose OSs around the world on unnecessary ticks is probably quite significant. This includes not only idle time, but also think time, as occurs for example between typing successive sentences. Note that CPU throttling has no effect on OS ticks, as these are performed regardless of any throttling activity.

To demonstrate the effect of ticks on energy consumption we connected a measuring power supply to a crippled laptop with all devices disabled. Somewhat surprisingly, the power used was not linearly related to the Hz (Fig. 4); instead, it achieved a min-

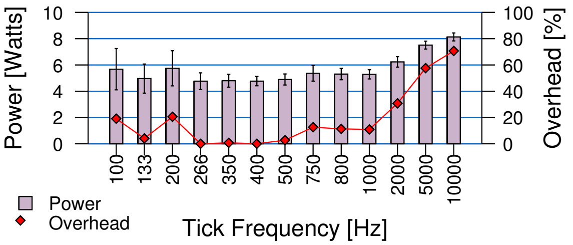

Figure 4: *Power consumption of an idle processor as function of Hz, and overhead measured as percent increase over the minimum of 4.77W.*

imum at mid-range (266–500Hz). Moreover, we observed that lower frequencies experienced two power states: some samples were as high as 7W, while most were around 4W. The solution to this mystery is that the idle loop executes the HLT instruction, which stops execution and places the processor in a HALT state until an interrupt occurs. At low tick frequencies, the processor has sufficient time to shutdown most of its units before an interrupt occurs, forcing it to restart those units. This shutdown/wakeup cycle is apparently power consuming.

When running an idle loop that executes a NOP instead of HLT, or when executing a simple loop, the power consumption is constant at around 14W regardless of Hz. The reason is that it does not matter whether the power is consumed by the idle loop, the user code, or the tick handler — they all require about the same. Thus the effect of ticks on energy is most pronounced when the system is idle. However, it should be noted that overhead of periodic timers also affects non-idle time, as it causes applications to run longer thus consume more power. In particular, there is a significant indirect cost of additional power-consuming cache activity after interrupts [9].

To conclude, it seems remarkable that such a huge effort is devoted to throttling (slowing the hardware clock when possible), while the useless activity of the OS clock is completely overlooked.

3 The Solution: Smart Timers

The typical solution to the problems described above has been to design a domain-specific alternative, for example, special schedulers to improve the support for soft real-time applications [1, 6, 11, 3]. But the root problem in all three domains is periodic ticks; if we can avoid them, we will have a more general solution. We therefore suggest that periodic timers be replaced with *smart timers*, defined here to combine three properties: **(1)** accurate timing with a settable bound on maximal latency, **(2)** reduced overhead by aggregating nearby events, and **(3)** reduced overhead by avoiding unnecessary periodic events.

A step in this direction was taken by *soft timers* [1]. These reduce the reliance on periodic timers by exploiting existing timing opportunities, such as each return from a system call (when the price of context switching to/from the kernel is already payed). Under some workloads such opportunities occur at a higher rate than ticks, thus improving average timing resolution. Unfortunately, the timing of a specific event cannot be guaranteed (violating the first smart timers requirement), so periodic ticks serve as a fallback (thus violating the third). Moreover, soft-timers are no help for the popular workload composed of a single soft real-time application (e.g. movie player) which goes to sleep between successive operations (frames display) while leaving the system idle (and therefore opportunity-less) in between.

A basic OS principle suggested by Finkel is that events at a high level are actually polling at a lower level [4]. For over 30 years OSs have used polling. It is now time to consider pushing it down to the hardware, leaving the OS to be event-based. This approach is adopted by *one-shot timers*, which are only set for specific events; when there are no pending events, the system simply relinquishes control, and allows a user application to make full use of the machine. One-shot timers have been used in several (mainly real-time) OSs [5, 8, 14]. However, without some bound on the interrupt frequency, this approach violates the second requirement of smart timers. Consequently, it is unsuitable for a general-purpose OS, as any user can effectively bring the OS down by generating millions of events with nanosecond differences.

The overhead for multiple finely-placed events is overcome by *firm timers*, which combine periodic, soft, and one-shot timers [6]. This mechanism uses an “overshooting” parameter S, such that if a timer is requested for time T, the system will actually set it to $T + S$. If the kernel is entered after T but before $T + S$ (e.g. due to a system call a-la soft timers), then the requested event will be executed and the associated overhead will be avoided. Otherwise, the timer will explicitly go off at $T + S$ and pay the overhead price — but amortize it by also firing all other timers requested for times up to $T + S$ (and set to actually happen up to $T + 2S$). Setting S to zero turns firm timers into one-shot timers.

Firm timers already meet two requirements of smart timers: allowing timing services as accurate as required (by setting S), and bounding the overhead of adjacent timers by aggregation. Divorcing firm timers from periodic ticks will turn them into full fledged smart timers. With such a mechanism, setting S to 100 μs will be similar in every respect to a 10,000 Hz kernel, save useless periodic ticks.

Fortunately, the three reasons given by Goel et al. [6] for choosing to build firm timers on the existing periodic tick infrastructure are not show stoppers. One is that keeping ticks allows for setting high resolution timers only for the next tick, which in turn allows for 32 bit operation; however, the alternative of 64-bit operation is available. Another is the claim that the data structures are more efficient, which may be true but is not crucial. The third is the desire to avoid breaking existing code that relies on the periodic work done every tick. We are currently in the process of turning Linux-2.6 to be tickless, and so far it seems the code explicitly executed on every tick can be dealt with in a straightforward manner: instead of billing the running process on every tick, this is done on each kernel entry, making billing and time keeping more accurate but costlier; instead of (SMP) load balancing on every tick, this is done upon every fork and exit; instead of running NTP code on every tick, this is only done when necessary, etc.

Context switching is also easily solved: if there are two or more runnable processes, a timer is set to end the quantum; otherwise, no timer is set, allowing the single process to run with no interference. The OS will resume control only upon other device interrupts (keyboard, mouse, network, etc.).

Another problem is the fact periodic ticks have been around for so long, some user code came to rely on them. Regrettably, the Linux HZ macro may be visible to users through inclusion of appropriate headers. For example, there are 13 occurrences of HZ within the source code of the Unix top utility. We hope that this practice is not widespread.

In conclusion, we note that current solutions for the problems related to OS ticks accept this polling strategy as if it was carved in stone: soft and firm timers build on ticks [1, 6], unaware of and conflicting with HPC noise reduction techniques which aspire to reduce Hz [7]; power conservation efforts only acknowledge ticks existence so that they will be taken into account [9]. This need not be the case. There is no technical limitation preventing a tickless OS (as one-shot OSs do in fact exist). The only question is whether this is realistic, in terms of existing software which can potentially be affected.

References

[1] M. Aron and P. Druschel, "*Soft timers: efficient microsecond software timer support for network processing*". *ACM Trans. Comput. Syst.*, Aug 2000.

[2] J. J. Dongarra et al., "*Top500 supercomputer sites*". URL http://www.top500.org/. (updated every 6 months).

[3] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS*, Jun 2003.

[4] R. A. Finkel, *An Operating Systems Vade Mecum*. Prentice-Hall, 2nd ed., 1988.

[5] E. Gabber et al., "*The pebble component-based operating system*". In *USENIX Annual Technical Conf.*, June 1999.

[6] A. Goel et al., "*Supporting time-sensitive applications on a commodity OS*". In 5th *Symp. Operating Systems Design & Implementation*, Dec 2002.

[7] T. Jones et al., "*Improving scalability of parallel jobs by adding parallel awareness to the operating system*". In *Supercomputing*, Nov 2003.

[8] I. Leslie et al., "*The design and implementation of an operating system to support distributed multimedia applications*". *IEEE J. Select Areas in Comm.*, Sep 1996.

[9] T. Li and L. K. John, "*Run-time modeling and estimation of OS power consumption*". In *SIGMETRICS*, Jun 2003.

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

[11] J. Nieh et al., "*A SMART scheduler for multimedia applications*". *ACM Trans. Comput. Syst.*, May 2003.

[12] F. Petrini et al., "*The case of missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q*". In *Supercomputing*, Nov 2003.

[13] J. C. Phillips et al., "*Namd: biomolecular simulation on thousands of processors*". In *Supercomputing*, Nov 2002.

[14] B. Srinivasan et al., "*A firm real-time system impl. using commercial off-the-shelf hardware and free software*". In *IEEE Real-Time Technology & App. Symp.*, Jun 1998.

[15] D. Tyrell et al. "*Rail passenger equipment crash worthiness testing requirements and implementation*". In *Intl. Mechanical Eng. Congress & Exposition*, Nov 2000.

[16] "*Linuxdevices.com*". URL http://www.linuxdevices.com/.