30 Seconds is Not Enough! A Study of Operating System Timer Usage

Simon Peter, Andrew Baumann, Timothy Roscoe
Networks and Operating Systems Group, ETH Zurich, Switzerland
{speter,andrewb,troscoe}@inf.ethz.ch

Paul Barham, Rebecca Isaacs
Microsoft Research, Cambridge, UK
{pbar,risaacs}@microsoft.com

ABSTRACT

The basic system timer facilities used by applications and OS kernels for scheduling timeouts and periodic activities have remained largely unchanged for decades, while hardware architectures and application loads have changed radically. This raises concerns with CPU overhead, power management and application responsiveness.

In this paper we study how kernel timers are used in the Linux and Vista kernels, and the instrumentation challenges and trade-offs inherent in conducting such a study. We show how the same timer facilities serve at least five distinct purposes, and examine their performance characteristics under a selection of application workloads. We show that many timer parameters supplied by application and kernel programmers are somewhat arbitrary, and examine the potential benefit of adaptive timeouts.

We also discuss the further implications of our results, both for enhancements to the system timer functionality in existing kernels, and for the clean-slate design of a system timer subsystem for new OS kernels, including the extent to which applications might require such an interface at all.

Categories and Subject Descriptors: D.4.m [Operating Systems]: Miscellaneous

General Terms: Experimentation, Measurement

Keywords: timers, kernel interface design, adaptability, scheduling

1. INTRODUCTION

This paper examines an area of OS kernels which appears to have received relatively little attention: the timer subsystem. The basic design of such a timer facility has remained fairly static for decades (compared, for example, to 6th Edition Unix [18]). Generally speaking, an OS kernel includes a facility to schedule a notification to a user-space task or kernel activity at some specified time in the future, possibly suspending execution of the calling task in the process. The interface to these systems (which we review briefly in Section 2) offers relatively simple, low-level functionality, and is used extensively by both the kernel and user-space applications.

However, we argue that it is time to reconsider how hardware timer functionality is presented by an operating system, both to applications and kernel subsystems.

Firstly, the increasing prevalence of networked applications has led to a considerable use of timer calls by applications, resulting in significant observed CPU overhead [4]. For this reason, the Windows Vista TCP/IP stack was recently completely re-architected to use per-CPU timing wheels [30] for TCP-related timeouts. Vista also dynamically adjusts the frequency of the periodic timer interrupt, processing timers according to observed CPU load.

Secondly, determining appropriate timeout values in such applications is also difficult and is generally left to the application programmer, leading to frequent arbitrary values (as we show in Section 4.2) and associated slow response times in the presence of failures. This problem can be compounded by layered software architectures such as user-interface code, resulting in heavily-nested timeouts.

Thirdly, timeouts with definite wakeup times can cause significant (and unnecessary) power consumption on systems that use low-power modes during idle periods. Such concerns have led to recent modifications in the timer interface provided by Linux [24]. Similarly, Vista delays periodic background jobs as a way to enable hard disk spindown for power saving.

Our motivations in studying how kernel timers are used in Linux and Vista are twofold. On the one hand, we want to assess what short-term enhancements to a kernel's timer functionality can have the greatest impact on user-perceived performance and responsiveness. On the other, we are interested in what a clean-slate timer system design for a new kernel might look like. This paper represents an early part of that design process.

The contributions of this paper are as follows:

- In Section 3 we present *options for instrumenting OS kernels* (in particular, Linux and Vista) to extract timer usage data. Since the timer primitives offered by these kernels are highly generic and low-level, simple logs of timer invocation convey very little information about how timers are being used. We present techniques and heuristics for obtaining a more complete picture of how timers are used by the kernel and applications.
- In Section 4 we present *measurement data and analysis* from studies of both Vista and Linux running under a variety of workloads. We show that the same timer subsystems in both kernels are used in at least five different ways. In addition, very few regular uses of timers are adaptive (in that they react to measured timeouts or cancelation times via a control loop), and many timers are set to "round number" values such as 0.5, 1, 5, or 15 seconds.

- In Section 5 we discuss the *wider implications* of our results for the design of a future timer subsystem from an OS kernel. In particular, we observe that much existing timer usage is closely related to OS task scheduling and dispatch. We consider whether a carefully designed CPU dispatcher along the lines of scheduler activations [2] might remove the need for user-space timer functionality entirely.

In the next section, we first set the context by summarizing the kernel timer facilities in Linux and Microsoft Vista.

2. TIMER SUBSYSTEM STRUCTURE

Timer subsystems in Linux and Vista provide very similar, and conceptually very simple, basic operations: a timer can be *set* or armed for some time in the future, an existing timer can be *canceled* before it *expires*, and an expiring timer is fired by calling the function closure associated with it. In addition, in some situations a thread or process can block on a timer, becoming unblocked when the timer expires.

The implementation of such a subsystem requires some form of priority queue for outstanding timers (typically implemented using a variant of timing wheels [30]), together with an upcall or interrupt handler from a lower-level timer. At the lowest level in the kernel, this sits above a hardware interval timer or periodic ticker, such as the (local) APIC in IA32 systems.

Such a timer subsystem is effectively a multiplexer for timers, providing a (potentially unbounded) queue of timers to its clients while requiring only a single timer (such as that provided by hardware) underneath. This allows subsystems to be "stacked" or layered, with each layer multiplexing timeouts onto the layer below. Timers in both Linux and Vista systems can be viewed this way, as a dynamic tree (or, on a multiprocessor, a forest) of timer facilities extending from hardware devices into application code.

In this paper we are primarily concerned with the broad semantics, usage, and structure of timer subsystems rather than their particular syntax, though we point out details where they are important.

2.1 Linux timers

Linux systems typically have two multiplexing layers, one in the kernel and one implemented as a `select` loop in the application, often in a library such as `libasync` [19] or Python's Twisted-Core [29]. While some kernel timer invocations are explicit (such as the timeout parameter to `select`), others are implicit: for example, a `write` call which ultimately invokes the hard disk driver will cause a command timeout to be installed.

The Linux kernel implements two independent low-level facilities: the standard timer interface, and the high-resolution timer facility. The standard timer interface is driven at a fixed frequency (the *jiffy*, by default 250Hz) by a hardware ticker device, such as the (local) APIC. Expiry times are expressed as an absolute time in jiffies since boot. Timers are represented as structures that are preallocated by clients of the subsystem, and delivery of timeout events is performed using simple callbacks outside of a process context (as part of a bottom-half handler).

The functions involved in modifying timers on Linux are:

- `init_timer`, `setup_timer`: Initialize a timer data structure. We refer to both as `init_timer` in this paper.

- `add_timer`, `__mod_timer`: Set a timer to a specified timeout value and set it running. We refer to both as `__mod_timer`.

- `del_timer`, `del_timer_sync`, `try_to_del_timer_sync`: Cancel a timer from the timer subsystem and set it inactive. The latter two functions deal with corner cases on a multiprocessor, and we refer to both as `del_timer`.

- `__run_timers`: Execute callback functions of outstanding expired timers. This function is called only from interrupt context.

Linux also defines higher-level functions for timer operations, some of which perform additional housekeeping tasks that are not of relevance to this study. The functions mentioned above are the most basic, and we will always refer to these.

Linux imposes few constraints on what functions can be called on a timer struct, once `init_timer` has been called – for instance, our traces show repeated deletions of an already-deleted timer. It is also common in the Linux kernel to reuse a statically-allocated timer struct for repeated timeouts.

Linux provides a facility for a thread executing in the kernel to block waiting for a timer, but (unlike in Vista) this is implemented by calling the timer system to install a callback, followed by a separate call to the scheduler to deschedule (block) the calling thread.

The large number of standard timers used in the Linux kernel have recently been identified [24] as a significant factor in CPU power consumption and utilisation: an otherwise idle CPU has to wake up frequently in order to serve expiring timers. This has led to a number of ad-hoc extensions to the basic timer interface:

- The `round_jiffies` and `round_jiffies_relative` functions, introduced in kernel version 2.6.20, round a given absolute or relative jiffy value to the next whole second. Timers that do not need to be precise about their expiry time can use these functions and will consequently time out in batches, reducing the overall number of system wakeups.

- The *dynticks* feature introduced in version 2.6.21 disables the periodic timer interrupt completely when the system is idle, allowing the CPU to sleep until the next event.

- Version 2.6.22 adds a *deferrable* flag to the timer subsystem. Timers that are marked deferrable function normally when the CPU is running, but will not wake up the CPU if the system is otherwise idle.

Unfortunately, these facilities are sufficiently recent that they are hardly[1] used in new kernels. We discuss their potential generalization further in Section 5. Later versions up to 2.6.23.9 (the version we instrumented for this study) did not change the standard kernel timer subsystem in any significant way.

Linux kernels from 2.6.16 onward also provide a second high-resolution timer facility [15] that is typically driven from CPU counters, although external clock sources can also be used.

From user space, only two system calls allow setting a timer without simultaneously blocking the calling process on some event: `timer_settime` and `alarm`. The former is part of the POSIX timer API and has a corresponding cancelation system call, the latter delivers an alarm signal after the specified timeout that can be canceled by another call to alarm with a timeout value of zero. All other system calls serve different purposes that only involve setting a timeout as a latest time of return from a long-running call.

[1]We found a total of 40 invocations of `round_jiffies` and `round_jiffies_relative` out of 1464 timer sets within Linux 2.6.23.9. The deferrable flag is used 3 times within that kernel.

2.2 Vista timers

Vista's timer facilities are considerably more complex. Timers are multiplexed at many more layers in both the operating system and applications. A significant difference with Linux (which makes it harder to trace timer usage throughout the software stack) is that most structures representing outstanding timers are allocated on-the-fly and not reused. Another difference is Vista's more generalist approach to thread synchronization. Many timer structures are subclasses of the basic synchronization object, meaning that threads can block on timers.

All of Vista's timer-related interfaces are ultimately implemented over the NT kernel's base `KTIMER` functionality. Kernel timers can be set for absolute times or relative delays and canceled using the `KeSetTimer`/`KeCancelTimer` interface and are added to a timer ring which is processed on clock interrupt expiry. Timers are a subclass of synchronization object and therefore threads can block on timer objects directly. Alternatively, expiry events can be delivered by deferred procedure call (DPC), analogous to Linux's bottom-half handlers. These timers are used extensively by device drivers and kernel subsystems.

Threads can block on any NT synchronization object via the `WaitForSingleObject` and `WaitForMultipleObject` calls, both of which accept an absolute or relative timeout parameter. Wait timeouts are implemented using a dedicated `KTIMER` object in the kernel's thread datastructure and have a fast-path insertion into the kernel timer ring. Thread sleep is also based on this mechanism.

Timer functionality is exported from the kernel via the NT API calls `NtCreateTimer`, `NtSetTimer` and `NtCancelTimer` which provide essentially the same abstraction but using asynchronous procedure calls (APCs) (analogous to Unix signals) rather than DPCs, and identifying timers via `HANDLE`s in the kernel handle table.

Recognizing that timers are used heavily in Windows, NT-DLL (which sits directly above the kernel interface) provides a user-level timer pool abstraction via the `CreateThreadpool-Timer`/`SetThreadpoolTimer` APIs. This is essentially a user-level timer ring multiplexed over a single kernel timer. Threadpool timers invoke a user-supplied callback function on expiry.

Above this, the Win32 API exposes timer functionality in two ways. Firstly, it provides the {`Create`, `Set`, `Cancel`}`Waitable-Timer` APIs, which expose the NT API interface largely unmodified. Secondly, it wraps these APIs in a form more suitable for event-driven GUI applications providing `SetTimer`/`KillTimer` calls. Timeouts from the kernel are delivered as APCs which insert `WM_TIMER` messages in the application's message queue. The queue is serviced by the message dispatch loop of the application's GUI thread.

As with Unix, timeouts are also supported via the Winsock2 `select` API. Unlike most Unix variants, these are actually implemented as a blocking `ioctl` on the `afd.sys` device driver, which allocates a *fresh* `KTIMER` object and requests a DPC callback at the appropriate expiry time to complete the `ioctl`.

2.2.1 User-space usage

Figure 1 demonstrates how often timers are used by applications and the operating system on a typical desktop machine. The graph shows the number of timers used per second by Outlook, Internet Explorer, system processes and the kernel over a 90 second excerpt from a trace. The kernel typically sets around a thousand timers per second, whilst a typical application such as a web browser will set tens of timeouts per second. Outlook uses around 70 timers per second when idle, but during bursts of activity can set as many as 7000 timers in a second. Upon investigation, this behavior was traced to a coding idiom whereby any upcall in user interface code is wrapped in a form of timeout assertion which catches upcalls lasting longer than 5 seconds.

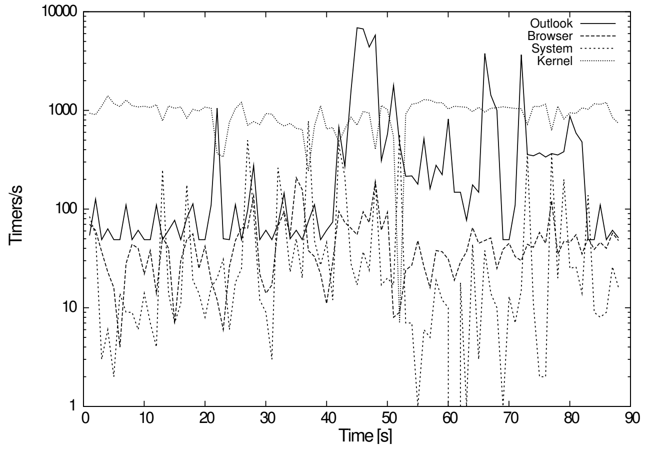


Figure 1. Timer usage frequency in Vista

2.2.2 The effect of layering

Large software systems are frequently composed of multiple layers or components which, though internally complex, interact through relatively simple and well specified interfaces. Successive layers usually provide increasing levels of abstraction, such as a file browser GUI over the top of a file system API over one or more file systems. Where components may be deployed in more than one configuration it is extremely difficult to design interfaces which allow internal performance optimizations such as caching. It is even more difficult to design interfaces which also handle failures and timeouts.

For example, on Windows, when the user types the name of a server into the file browser, parallel name lookups are initiated using WINS, DNS and other name providers. The application must wait for one or more of these to succeed and sets timeouts on each alternative. Typing an incorrect name can frequently result in a long wait. Assuming name resolution is successful, the browser will next attempt to connect to the server using a variety of network file system protocols including SMB, NFS and WebDAV, again these alternatives are tried in parallel with timeouts. In the case of NFS (implemented over SunRPC) many implementations respond to refused connections with an exponential backoff which retries 7 times, doubling the initial 500ms timeout each iteration. Thus, recovering from a typing error can take over a minute!

RPC subsystems are often designed to run over unreliable transports so individual operations have their own timeouts—often defaulting to a few seconds. Nowadays RPC is usually configured over a TCP layer that again involves a complex series of timeouts, both static timeouts for connection setup retries and adaptive timeouts for packet retransmission due to loss.

In this seemingly simple example, it is not surprising that when network connectivity to a file system is lost it can take tens of seconds to present this failure to the user, and that the underlying cause is lost in the process. Although a response from the file server usually arrives shortly after the 130ms round-trip time, the increasingly conservative layered timeouts cause the operating system to take much longer to notice the error than the end user.

3. METHODOLOGY

In this section we describe our methodology for logging timer activity and the workloads under which we took traces.

As was discussed in Section 2, each system layer multiplexes timer activity onto the layer below. Therefore, a trace of timer calls at a low layer, such as in the kernel's timer code, while giving good coverage of timer usage throughout the system also makes it difficult to identify the true sources of timers. Furthermore, a low-level instrumentation point masks the distinction between a single timer whose value varies and multiple timers that are being coalesced and thus appear to come from the same piece of code. To distinguish such timers, as well as to identify the subsystems or applications that use timers, we log stack trace data, process information and timer structure addresses, in addition to timeout values.

Since Linux and Vista are quite different systems from an instrumentation point of view, we describe each separately, starting with Linux.

3.1 Linux instrumentation

Linux already includes functionality to collect timer statistics as part of the kernel debug code, providing a rough estimation of timer usage in the Linux kernel. However, in order to observe the details and duration of different timers, additional information needs to be observed. Thus, we implemented separate functionality to log all calls to the timer interface, including setting, cancelation, and expiry of timers. In Linux the stack trace enables us to find out the exact part of a kernel subsystem that registered a timer. For example, since the TCP implementation is part of the IP subsystem and uses its functions to register timeouts, recording only those functions which register timers is insufficient to identify the originator as the TCP stack. For user-level timers, although we use the process ID and call stack to identify the timer, we cannot currently identify the sources of timers beyond the system-call level.

A problem we encountered in the Linux tracing concerns conversions between absolute and relative timeouts within the kernel. We measure the event of arming a timer in the kernel's `__mod_timer` function, which accepts the absolute time value in the future at which the timer will expire. This value is computed earlier by the kernel code, usually as a relative calculation from the current time. Since program execution takes time as well, and in some cases can be pre-empted by another thread, the observed timeout values exhibit jitter. Our classification of kernel-space timeouts accounts for this jitter by allowing a variance of up to 2 milliseconds in the observed timeout values. This value was experimentally determined from observations of the kernel work-queue timer, a timer with a known fixed period. No such jitter occurs in case of user-space timeout values, as only relative values are accepted by the system calls that arm timers and we measure these values directly at the system call.

3.2 Linux environment

We chose to run our experiments on real hardware, and used low-overhead binary logging to gather trace information. An alternative approach that is feasible for certain workloads and which we explored in an earlier version of this paper uses a virtual machine environment, for example the QEMU emulator [8]. Virtualization has a clear benefit: it permits running complete operating systems without modification, while allowing the virtual machine to be stopped and analyzed at any time using an attached debugger. This setup provides a completely unintrusive experimentation environment. However, for workloads involving entities not under control of the virtual machine monitor, such as our network workload, virtualization cannot be used without perturbation of measurement results. Also, time is notoriously hard to virtualize. Thus, the accuracy of measured timing information within an emulator may be doubtful.

Paravirtualization, as offered by Xen [6] or User-mode Linux (UML) [13], is less attractive for this work, as it changes architectural details that would perturb measurement results. For example, UML is implemented as a separate architecture inside the Linux kernel with its own device drivers. As many kernel timers are architecture-specific, we would obtain different results from UML than a purely emulated system.

We ran our experiments on a PC with 8GiB of RAM and Intel Xeon X5355 processors running at 2.66GHz, connected to the department's LAN using a Gigabit Ethernet interface. The system ran in 32-bit mode on a single processor. The LAN is routed to the Internet.

For all experiments, we use a base Debian 4.0 operating system, running the Linux kernel version 2.6.23.9, which has been compiled to include debug symbols and frame pointers. All other options are kept at their defaults (note that our kernel is therefore configured without kernel preemption).

We developed a logging system that uses the relayfs [33] high-performance binary logging infrastructure within Linux to log data into a 512MiB buffer in kernel memory. We ensured that all of our traces were able to fit into this buffer. After running the workload, we used a user-space program to read out the buffer and convert the trace into a textual format, which we then processed to gain the results presented in this paper. relayfs ensures ordering of logged events, and that new events cannot overwrite old logs.

We conducted several benchmarks to measure the impact and overhead of our logging scheme:

- A micro-benchmark of the code executed to gather required timeout parameters and log these to the memory buffer shows an overhead of 236 CPU cycles. This result was calculated by measuring the time for 1,000,000 consecutive runs of our measurement code.
- We ran a timer-intensive workload, once with an unmodified kernel and once with our logging enabled, and used the cyclesoak [20] program to determine the respective aggregate CPU idle time available during each run. The difference is within 0.1% of total CPU overhead.
- We counted the overall number of calls to the timer subsystem for the same workload on both an unmodified kernel and with our logging enabled. The difference is within 3% of the number of calls.

3.3 Vista instrumentation

In order to capture all timer-related activity on Vista, it was necessary to instrument both the `KTIMER` interface and the thread wait codepaths. Similarly to relayfs, the event tracing for Windows (ETW) facility [22] provides extremely low time and space-overhead kernel-mode logging and proved to be ideal for our purposes. We added events to the `KeSet-` and `KeCancel-` timer calls, and also to the clock interrupt expiration DPC that processes the timer ring and fires timeout DPCs.

As described in Section 2.2, thread wait primitives have a fast-path timeout code, and so required explicit instrumentation. We added a single event on thread unblock which logs the timestamps before and after blocking, the user-supplied timeout parameter, and a boolean indicating whether the wait was satisfied or timed out.

The problem of identifying repeated instances of the same high level "timeout" is endemic in Vista. For example, several common codepaths allocate kernel timer objects dynamically, and hence repeatedly calling `select` on the same socket will not typically result in operations on the same kernel timer. Similarly, the thread wait primitives accept a user-supplied set of synchronization objects which may or may not be constant across invocations from the same call-site. However, the Vista instrumentation is able to capture both the kernel- and user-mode stack for each log event, which can be post-processed to cluster operations according to call-site and thread ID and to help identify cross-layer timeout interactions.

3.4 Vista environment

We ran our experiments on a Dell Precision 380 Workstation with an Intel Pentium D CPU running at 2.8GHz, with 2GiB of RAM, connected to the LAN using a Gigabit Ethernet interface.

The test machine was running Vista Ultimate Edition 32-bit, however it was booted with a privately-compiled kernel that contains our additional instrumentation—although extensive ETW instrumentation already exists throughout Vista, this study required the addition of four custom events to the kernel.

3.5 Workloads

Based on initial observations of timer usage, we identified four workloads to drive our experiments:

1. an idle desktop system,
2. the Firefox web browser displaying a page from myspace.com,
3. the Skype internet telephony program making a call,
4. a web-server under load.

The Idle, Firefox, Web-Server and Skype workloads run for exactly 30 minutes. We justify this length by the observation of common timeout values: apart from the 7200 second TCP keepalive timer shown in Figure 3, whose behavior is well understood, there are no significant timeout values greater than 1000 seconds. This allows us to observe the lifetime of all significant timers.

The Linux idle system consists of the Debian base installation running the X window system and a window manager (icewm). The system has already booted, and stock system daemons such as syslogd, inetd, atd, cron, as well as the portmapper and gettys, are running. The system is connected to the network, but no network accesses from the outside are happening. We use this workload to gain an overview of timer usage within a typical idle desktop system.

The Vista idle workload consists of a standard Vista desktop install, with a user logged in on the console. No foreground applications were started, but 26 background processes (in addition to the System and Idle tasks) were running.

For the Firefox workload, we ran Firefox version 2.0.0.6, displaying a webpage[2] that makes use of the Macromedia Flash plug-in and JavaScript. No user input was provided while this workload is measured.

For the Skype workload, we installed Skype version 1.4.0.99 and directed it to make a phone call to another Skype account.

[2]The URL used was http://www.myspace.com/barrelfish.

We installed the stock Apache 2.2.3 web-server on the test system and the httperf [21] benchmark on another machine connected to the LAN to drive the webserver workload. For the Linux experiment, the LAN ran at gigabit speed, but for the Vista experiment, a 100Mb switch was used between the server and the client. Httperf is able to simulate artificial workloads and can replay pre-recorded real workloads. We set it to generate an artificial workload of 30000 HTTP requests, emitting 10 parallel requests at a time. Each request is encapsulated in its own connection, and connections use a timeout of 5 seconds on each state before they are considered broken and canceled. On Linux, the X window system was not running during this workload.

4. RESULTS

In this section, we present a series of salient results from our data.

Table 1 shows a summary of our Linux data: *timers* shows the total number of allocated timer data structures in each trace, *concurrency* the maximum number of outstanding timers at any time, *accesses* is the total number of accesses to the timer subsystem, and *user-space kernel* show the number of explicit and implicit accesses from user-space and the kernel. *Set*, *expired*, and *canceled* show the total number of operations of each type during the trace. Table 2 shows a similar summary for Vista.

	Idle	**Skype**	**Firefox**	**Webserver**
Timers	47	74	95	103
Concurrency	25	32	36	31
Accesses	165345	535686	3948490	283634
User-space	148603	517291	3927194	77272
Kernel	16742	18395	21296	206362
Set	63183	198021	1401976	112998
Expired	36477	65883	262703	19518
Canceled	26835	132553	1140744	96006

Table 1. Linux trace summary

	Idle	**Skype**	**Firefox**	**Webserver**
Timers	144	219	228	135
Concurrency	75	80	84	73
Accesses	270691	2169896	5202502	275786
User-space	55771	1424791	4924561	72476
Kernel	214920	745105	277941	203310
Set	252178	2101677	5186065	259871
Expired	233489	2016165	5054879	242775
Canceled	18659	68438	16665	16050

Table 2. Vista trace summary

We can see that timer usage is extensive in both the kernel and applications, and that GUI applications can be responsible for a very large number of timer calls. We can also see that on Vista timers more often expire, whereas on Linux more timers are canceled.

4.1 How are timers used?

A natural question to ask is what the timer facility is being used for. In principle this question can be answered simply by code inspection, though the multi-layered complexity of the timer system on Vista makes this all but intractable. Even on Linux this task presents a considerable challenge when considering the entire software stack from the low-level drivers up to the X window system and a complex application such as Firefox.

In any case, such an analysis is unlikely to yield information on which timers are used frequently. Since our motivation for this work is both to improve existing timer systems and design a new one, we are interested in the use-cases which are most common at runtime, not necessarily those that occur most frequently in the code. It is the former where improvements to a timer system should be expected to show the greatest impact.

Hence we adopt a hybrid approach: we profile running systems, identify patterns in timer usage, and then inspect code paths. This leads us to a useful taxonomy of how the timer subsystem is used in practice.

A natural criticism at this point is that surely such a taxonomy is obvious to the developers of the operating systems. We claim that reality is not so clear-cut: OS developers use timer facilities to solve particular problems at hand, and may not have the luxury or inclination to reflect on wider classes of usage—they focus (with good reason) on point solutions.

Moreover, given the propensity of systems hackers towards abstraction, one would expect to see such a taxonomy enshrined in programming constructs or functions, or at least discussed at length in kernel documentation or design papers. We find no such documentation, and the timer primitives are generally invoked directly regardless of usage mode (a notable exception is Win32, which has programming constructs for timeouts). Consequently, we consider it a valuable exercise to classify actual programming practice into usage patterns, and correlate these patterns with trace data showing invocation frequencies.

4.1.1 Usage patterns

Since many timers are always set to constant values, we consider such timers first. As before, we start with Linux, since the reduced layering and the widespread static allocation of timer structures makes it easier to correlate successive uses of a timer.

We observe that a timer used repeatedly shows one of several patterns of behavior:

- The timer always expires, and is immediately re-set to the same (relative) value. This corresponds to a *periodic* ticker, such as the Linux page out timer. A variant of this is the behavior seen with the X server in section 4.2, where a `select` timer repeatedly counts down until being reset to its former value.

- The timer never expires: before its expiry time, it is re-set to the same relative value in the future. This is a *watchdog*: it is endlessly deferred unless some (presumably) serious system condition prevents deferral. An example is the Linux console blank timeout.

- The timer usually or always expires, and after some (non-trivial) time interval, is set again to the same time value. Threads *delaying* execution for a fixed interval show this pattern.

- The timer almost never expires but instead is canceled shortly after being set, and after some (non-trivial) time interval, is set again to the same time value. This is a *timeout*, e.g. RPC calls or IDE commands.

Since re-setting a timer takes time as well, we allow a variance of 2 milliseconds between a timer expiry and its subsequent reset when classifying these timers. This value has been experimentally determined as mentioned in Section 3.1.

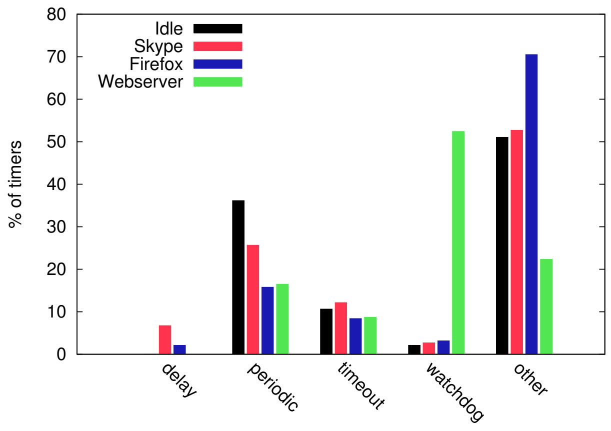


Figure 2. Common Linux timer usage patterns

Figure 2 shows the frequency of these use-cases in our Linux traces. We can see several expected features: for example, Apache uses watchdogs to timeout connections, whereas the Idle workload employs almost none, but is instead dominated by periodic background tasks.

The high number of unclassified timers in the Skype and Firefox workloads correspond to a large volume of very short timers: 4, 8 or 10ms, or 1, 2 or 3 jiffies. Both are soft real-time tasks (Skype is processing audio, and Firefox is playing Flash animations), and we conjecture that the minimal timer settings are an attempt to create a soft real time execution environment over a best-effort system, a question we return to in Section 5.5. The high number of unclassified timers in the Idle workload is due to an idiomatic use of `select` by X and the window manager which we examine below in Section 4.2.

Vista traces are also marked by a significant proportion of periodic and watchdog timers, and show a further distinctive pattern:

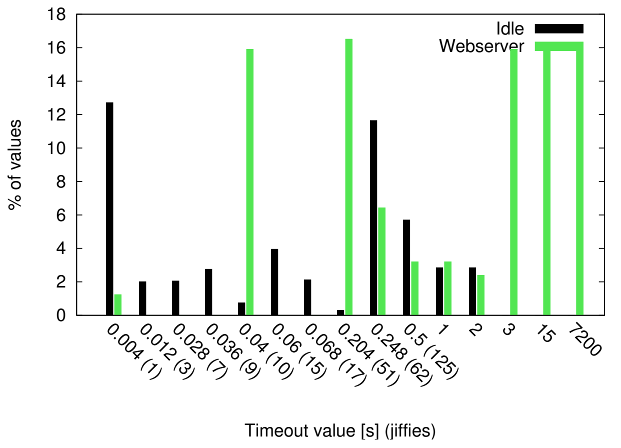
- The timer is repeatedly deferred by a constant amount each time as with a watchdog, but after a few iterations expires, before being restarted again. This mode is used for a deferred operation, for example lazy closing of handles to Vista registry contents. The idea is that the expiry triggers an action which should be taken when the activity in question has been idle for some period.

A final "pattern" is hard to recognize except by elimination, but is frequently seen in `select`-like uses where clients have multiple outstanding timers and are in addition waiting for thread synchronization on multiple objects. In this case, there can be very little regularity in either timeout values, or whether they are subsequently canceled or expire. However, in these cases the timer is usually reinstalled soon after expiration or cancelation—indicating an iteration of the client's event loop.

4.2 Commonly-used values

The most immediately apparent finding from our data on both Vista and Linux concerns the distribution of the timeout values when timers are set. Figure 3 shows the frequency of each timeout value responsible for more than 2% of the timers in each trace in Linux. Collectively, these timeout values represent 32% of all timers set during the traces.

For the Webserver workload, of which 97% of the timeouts are shown, it is evident that many timeout values in Linux have been



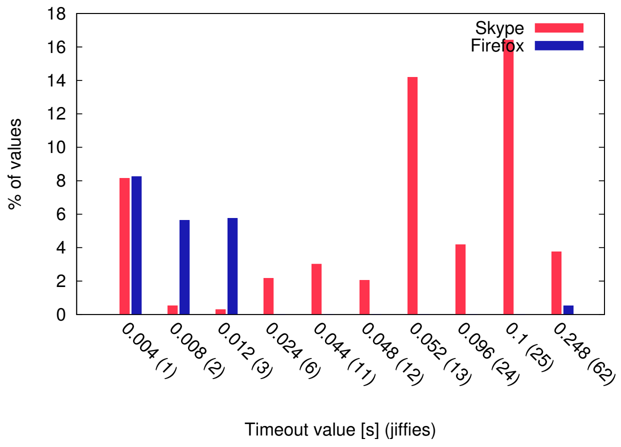


Figure 3. Common Linux timer values ($\geq 2\%$)

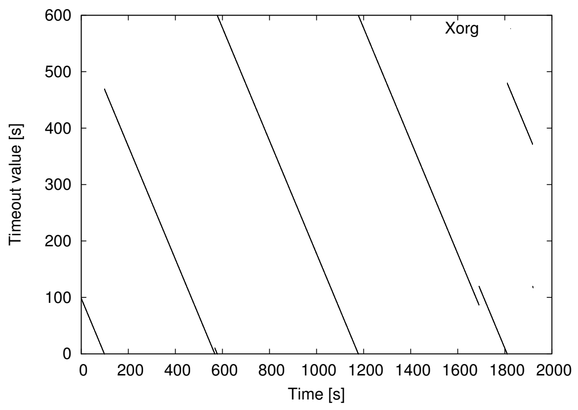


Figure 4. Dot plot of X timer usage via `select`

Timeout [s]	Origin	Class
0.004	Block I/O scheduler	Timeout
	Firefox polling file descriptors	Timeout
0.008	Firefox polling file descriptors	Timeout
0.012	Firefox polling file descriptors	Timeout
0.04	Sockets	Timeout
0.052	Skype	Other
0.1	Skype	Other
0.204	TCP retransmission timeout	Timeout
0.248	USB host controller status poll	Periodic
0.5	High-Res timers clocksource watchdog	Periodic
1	Kernel workqueue timer	Periodic
	Apache event loop	Timeout
2	Kernel workqueue	Periodic
	ARP	Periodic
	e1000 Watchdog Timer	Periodic
3	Sockets	Timeout
4	ARP	Periodic
5	Dirty memory page write-back	Periodic
	`init` polling children	Periodic
	Packet scheduler	Periodic
	ARP	Timeout
8	ARP cache flush	Periodic
15	*apache2* socket poll	Timeout
30	IDE Command timeout	Timeout
7200	TCP keepalive	Timeout

Table 3. Origins and classification of frequent Linux timeout values

determined offline and by human beings, rather than online and/or by machine calculation. Only some timeouts, like 0.204, which is the TCP keepalive timeout, are determined by online adaptation.

At first, the story seems different for the Idle (56%), Firefox (21%) and Skype (62%) workloads, until we examine the traces more closely. Figure 4 shows a small portion of the Firefox trace. Both the X server and the `icewm` window manager start by setting a constant timeout for `select`. When `select` returns due to file descriptor activity, Linux updates the timeout value to reflect the time remaining, and the processes use this new value until it reaches zero.

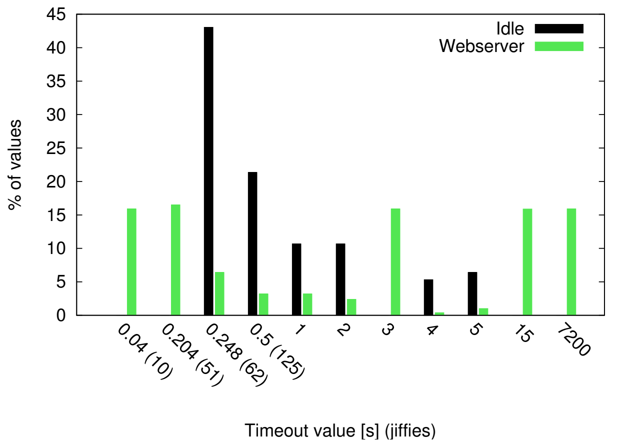
If we take this behavior into account (see Figure 5), it is clear that also for the Idle workload, almost all timeout values in applications and the kernel are determined by the programmer at compile time. Since this behavior, which is dominated by the X server and the window manager, distorts our measurement results with a well understood pattern, we filtered timers allocated by these programs from the results presented in all following figures. Firefox employs the same mechanism, seen in Figure 5 as a countdown from 3 jiffies. The only slightly more adaptive application is Skype, setting a number of short, irregular timeouts using `poll` and `select` throughout its workload, but is dominated by constant timeouts of 0, 0.4999 and 0.5, as shown in Figure 6.

In Linux we see a high correlation between timeout values and the static addresses of timer structures. This allows us to create Table 3, which shows a detailed list of the origins of these frequent timeouts within the kernel. Most are self-explanatory. In the table, we also list constant timeout values that do not occur frequently ($< 2\%$) in our traces, but are nevertheless interesting. We also left out values belonging to adaptive timeouts.

Human time-scales also dominate when we consider only timeouts set from user-space, as shown in Figure 6.

Figure 7 shows the results for our Vista workloads, which are similar. It is much harder to correlate such values with usage, but it is clear that a similar thing is happening in the Idle workload (in which 91% of timeout values are shown) and Webserver workload (86%), and to a lesser extent in the Skype (46%) and Firefox (13%) workloads.

The message of these results is that most timers are always set to a fixed, programmer-decided value or (as with `select`) repeatedly count down from such a value, and are not derived from measuring the system at hand. We return to this issue later.



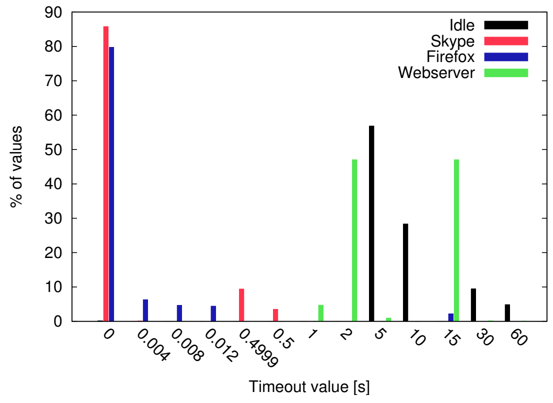
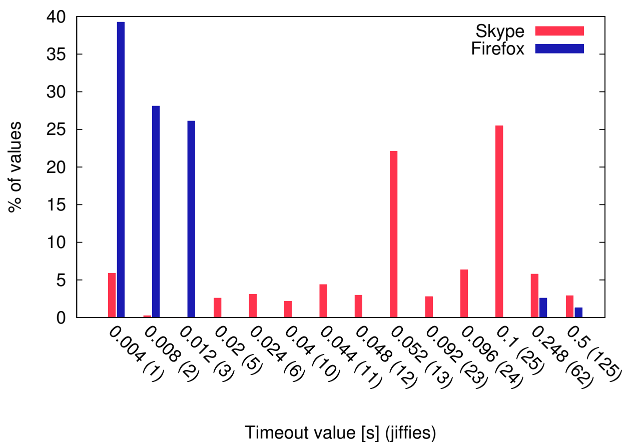


Figure 6. Common Linux syscall timer values ($\geq 2\%$)



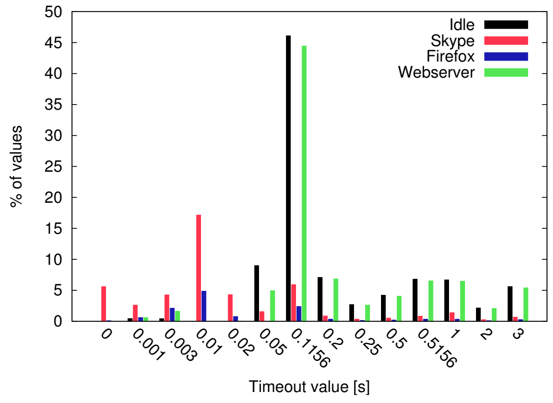


Figure 7. Common Vista timeout values ($\geq 2\%$)

Figure 5. Common Linux timeout values ($\geq 2\%$), filtered from X and icewm

4.3 Observed timer durations

We have shown that most timers are set to fixed values, but we have not yet considered when they expire or are canceled. In this section, we examine for how long timers actually run. Figures 8–11 plot for each workload the value each timer was set to versus the percentage of this time after which it was canceled or expired. The size of a circle represents the aggregate value frequency. Timers set to expire immediately or with an expiry time in the past are not plotted. As in the previous section, we filtered the X and icewm `select`-loop timers from the Linux results.

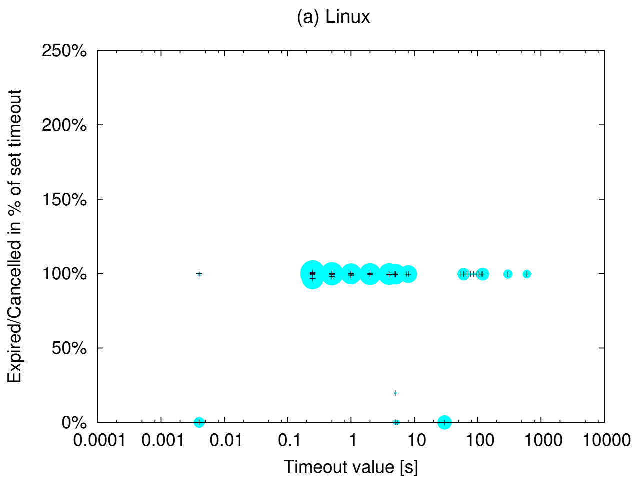
Many points on these graphs lie above 100%; this is an indicator that the timer expiry was delivered sometime after the scheduled time—particularly the case with short timeouts that are close to the system's scheduling granularity. The plotted points for timeouts shorter than around 10ms tend to exhibit a hyperbolic curve, due to the logarithmic time axis and the roughly-constant time required to set and then deliver a timer expiry notification. The figures are cut off above 250% (timers that expire more than one and a half times later than their set expiry time)—this happened mainly with very short timeouts in the Vista traces.

The many timeouts set to very short values have dubious value as useful timekeeping functions—most are delivered a significant fraction of their duration *after* their expiry time. This is understandable given the structure of the operating systems, but it is interesting that application developers still employ them. In particular, the Vista Firefox trace shows timers of this kind, having a timeout of less than one millisecond and being delivered at essentially random times.

Linux rounds timeouts to the nearest jiffy. Therefore, we do not see any timers of less than one jiffy (4ms) in the Linux traces, and there is a quantization effect for small timeout values which is not seen in the Vista traces.

Looking at specific workloads, we first see that in the Idle workload on Linux, most timers expire at the set time, and a few are immediately canceled. However, on Vista there are many more timeouts, both small and large, being set and delivered at variable delays. Common sources of timers on the idle Vista system included the *Idle* and *System* tasks themselves, the user processes *csrss.exe* and *svchost.exe*, and a third-party system tray application for the audio device, all of which set more than two timers per second.

The large cluster of points below 1 second seen in the Skype workload is characteristic of adaptive timers and is indeed an irregular pattern emitted by the Skype program through the `select` and `poll` system calls, as well as by adaptive socket timers. The array of points up to 50% at 3 seconds originates from socket timers. Linux's rounding of timeout values to the nearest jiffy is clearly visible here. The five second timer, which also appears in the Linux traces of the Firefox and Webserver workloads, originates from the ARP code within the Linux kernel. This code sets a constant five second timeout that is canceled at random intervals after it has been set. We account this to activity on the LAN that is part of our test environment. On Vista, the Firefox workload uses an even larger number of timers (2881 timers are set per second), many well below 10ms.



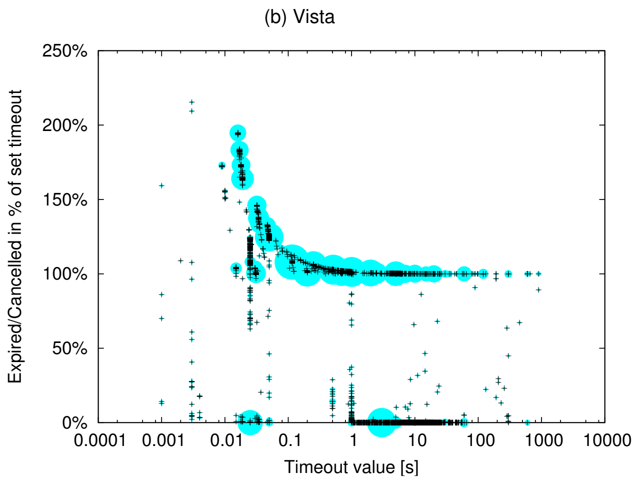
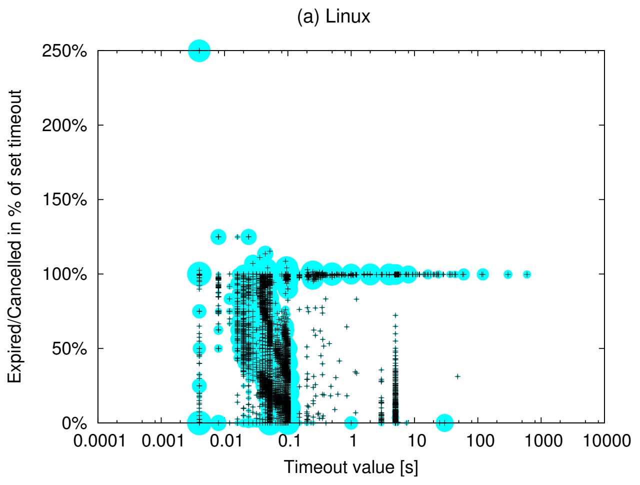


Figure 8. Aggregated timeout expiry/cancelation times correlated with timeout duration (**Idle** workload)



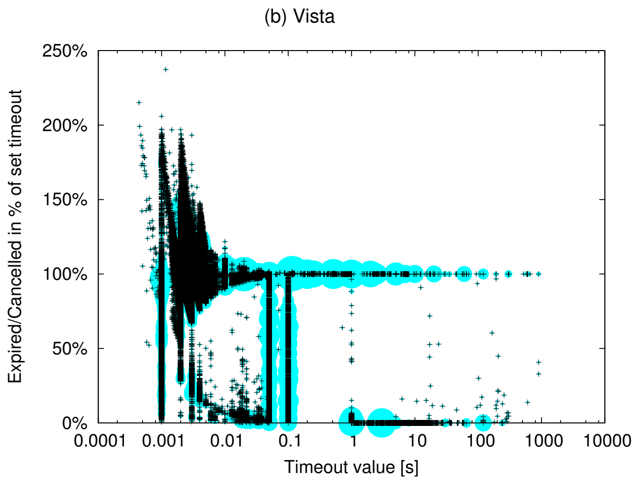
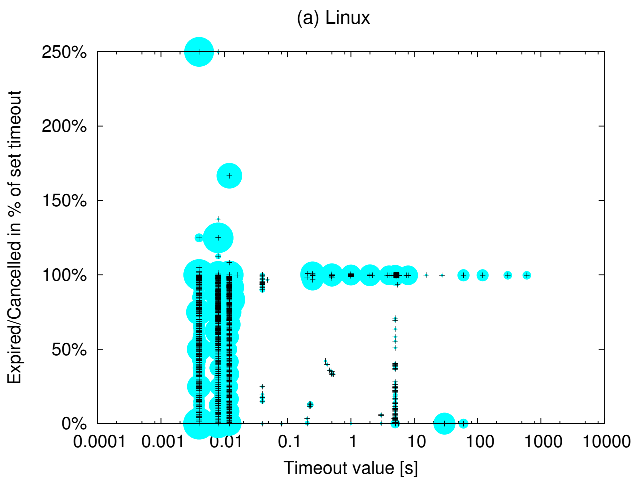


Figure 9. Aggregated timeout expiry/cancelation times correlated with timeout duration (**Skype** workload)



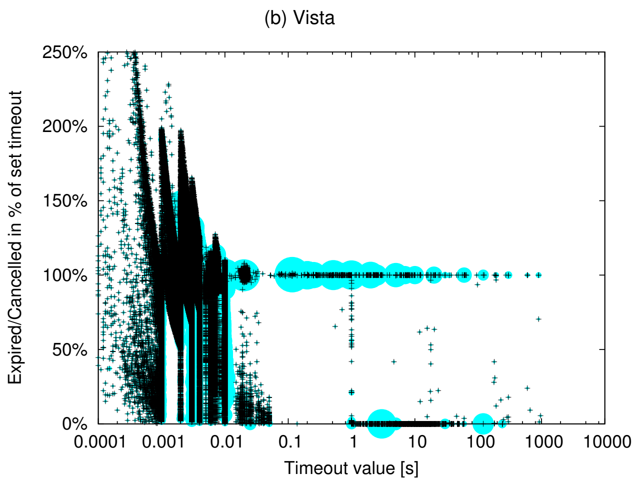
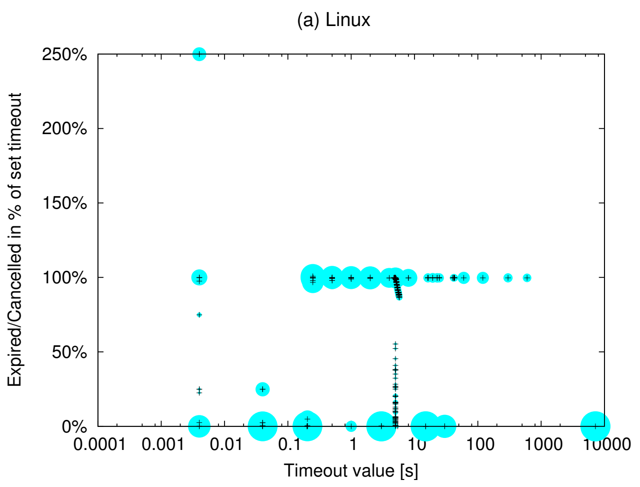


Figure 10. Aggregated timeout expiry/cancelation times correlated with timeout duration (**Firefox** workload)



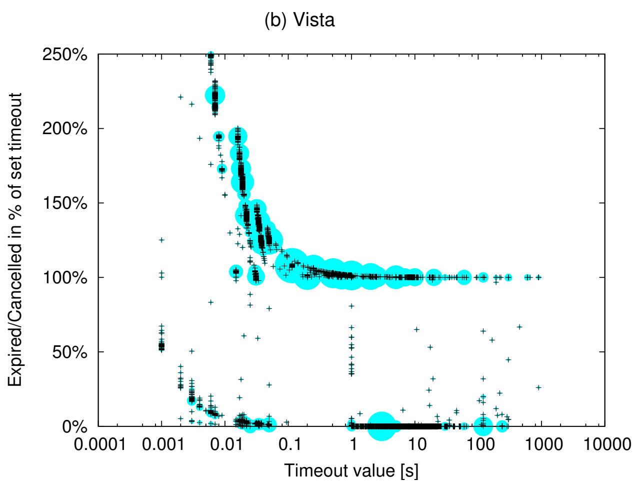


Figure 11. Aggregated timeout expiry/cancelation times correlated with timeout duration (**Webserver** workload)

The cluster of points between 80% and 100% around 5 seconds in the Linux Webserver workload is due to timers in the filesystem journaling code that already have adaptive timeout values and are mostly canceled. The Webserver workload on Vista appears similar to the Idle workload on the same OS, and interestingly does not include the 7200 second TCP keepalive timer that is used by Linux.

If frequent timeouts amenable to online adaptation existed, we would expect to see them as a vertical array of circles with some at or above 100% (representing expired timers), but a significant number well below 100%, and a large gap in the higher percentages in between. In the case of responsiveness, we are only interested in timeout values above 0.5 seconds, since below this the effect of adaptation may not be noticeable—though in a tight control loop like TCP this may still be important. Such an array can be seen in the Linux figures at 5 seconds, however as we have explained, its variance is due to network activity on our LAN.

The Skype and Firefox traces show clear cases of timeouts which are mostly (but not all) canceled well before they expire, however, these are all timeouts of very short duration. Adapting timeouts at this short timescale is not very helpful to responsiveness, although it reduces the number of system calls and thus kernel crossings as the cancelation of timers is equally distributed between 0% and 100%, showing that the calls actually block Firefox for a short amount of time. Given the sheer number of timer subsystem accesses in the Firefox workload, timeout adaptation would significantly decrease this overhead.

5. IMPLICATIONS AND DIRECTIONS

We embarked upon this study for two reasons: firstly, we had an intuition that existing timer subsystems were suboptimal in operating systems in use today and wanted to verify our hunch, and secondly, we wanted some empirical results as a basis for both enhancing existing OS timing facilities and designing a new one.

From a historical perspective, applications, workloads, and machine capabilities have certainly changed since the design of the Unix timer system, while the basic set/cancel timer interface has remained essentially unchanged [18]. While its survival is a testament to minimalist elegance, recent changes and our findings here both indicate that a rethink may be in order.

In this section, we conjecture about a set of features that would result in a more flexible timer system that would lend itself better to dynamic optimization of CPU usage, power consumption, responsiveness, or some tradeoff thereof. We start with modifications and additional functionality that our results suggest would benefit Linux and Vista, and then present more general design principles for a combined CPU scheduling and timer system which would finesse most of the issues we have encountered in our measurements.

5.1 Adaptive timeouts

As we have seen, timers are usually set to fixed, hardcoded values, but they could instead be adaptive. A computer system learns a great deal of information from its environment that can, and we believe should, inform selection of suitable timeout parameters at runtime.

A prominent example of the use of adaptive timeouts (as well as fixed ones) is of course TCP [16, 27], which constantly maintains a reasonable value for its retransmission timeout that is based on network conditions. It monitors the mean and variance of round-trip times and uses these to adjust the timeout value. When packets are lost or delayed, TCP avoids exacerbating the problem by applying an exponential backoff algorithm to increase the retransmission timeout value on each iteration. TCP's retransmission timer is an instance of using a timeout to detect failure and to improve responsiveness. The protocol also incorporates a persistence timer that is used to detect deadlocks that would otherwise occur when an acknowledgment packet is lost and both ends of the connection are left waiting for the other. This timeout determines the frequency with which the sender probes the receiver, and like the retransmission timeout, is adjusted using exponential backoff up to a maximum.

One might generalize this. For example, when a programmer begins to wait for a network message to arrive, rather than specifying a willingness to wait for an (arbitrary) 30 seconds, the programmer should request to "time out" once the system is 99% confident that a message will never be arriving. Note that this does *not* mean that failures will be detected 1% of the time. The confidence interval can be calculated by learning the distribution of wait-times for each timer object. The question is whether it is feasible to fit a simple model to the distribution of wait-times in a running system.

Having a model correctly increase and shorten wait times clearly entails modifying the timer system in the OS to continue monitoring for the event that was being waited for. There is a further challenge, however: wait times might increase due to a change in environment. For example, a user who normally accesses a network file system via a local-area network will see very different latencies if they try to access the same network file system from a wide-area network connection while traveling. Other causes of wait time changes might be workload changes causing performance variation. Frequent changes in latency will be reflected in our learned confidence intervals, but sudden and long-lived level shifts in latency will cause the whole learned distribution to shift.

5.2 Timeout provenance and dependency

Tracking timers effectively, particularly across the abstraction barriers imposed by `select` loops and other timer multiplexers, is challenging in current systems. There are clear benefits to be gained from preserving and propagating information about how timers have been set, and by whom, throughout the system. While it would have made our lives easier in correlating timer setting and expiry with kernel- and user-space activities, this has wider applicability.

In particular, with the widespread use of timers in application programming (particularly over Vista, but also increasingly with event-driven user-interface toolkits over Linux), debugging complex systems that internally employ watchdogs, timeouts, and periodic tasks will be a serious challenge. There are clear parallels here with the labeling of requests in multi-tier applications: being able to trace execution through the system is a critical requirement for understanding anomalous behavior [5, 10, 11]. We return to the more strategic implications of this in Section 5.5 below.

Ideally, explicitly capturing timer provenance information should go beyond single timers, or a collection of independent timers multiplexed onto a lower-level facility. Timers do not always stand on their own. Common idioms we have seen in GUI programming suggest that timeouts are frequently nested—operations that time out at one layer are retried until a higher-level, enclosing timeout fires. Other such dependencies are possible; in theory, the following relationships between two timers t_1 and t_2 can be identified:

1. t_1 overlaps t_2: This is the case when t_1 is set prior to or at the same time as t_2 and its expiry time is later than that of t_2. Overlapping timers waiting on the same event can be further classified:

   (a) Either just t_1, or both t_1 *and* t_2 expiring signify a failure of some kind. In this case $\max(t_1,t_2)$ is the expiry time and we may not need t_2. An example of this situation occurs in Section 4.4.5 of the DHCP specification [14].

   (b) Only t_2 need expire to signal a failure, in which case $\min(t_1,t_2)$ is the expiry time and we may eliminate t_1.

   (c) Neither t_1 *nor* t_2 need expire. In this case the only thing we can do is to cancel the other timeout, as soon as one of them is canceled. An example is the TCP keep-alive and retransmission timers, described in Subsection 4.3.

2. t_2 depends upon t_1: In this case, t_1 is set first and t_2 is only set upon cancelation/expiry (depending on the relation) of t_1. Periodic timers are self-dependent by this definition.

Inferring, or allowing programmers to explicitly declare, such relationships between timers would not only lead to better traceability and debugging, it would allow the timer implementation to optimize and adapt their behavior.

In practice, overlapping and dependency relationships are interchangeable: an overlapping relationship can always be transformed into a dependency relationship and vice-versa. For example, assuming that t_1 overlaps t_2, we can first set t_2 only and upon its expiry, we set t_1 only for the remaining time. If t_2 is canceled, we might not need to register t_1 at all, depending on the overlapping relationship - one technique to reduce the number of concurrent timers.

The use of layering and information hiding principles in object-oriented design provides problems for timer-based systems. One example we have shown is where naïve layering of timers leads to suboptimal behavior in the presence of failure (Section 2.2.2). A second example (see Figure 10) is where a large number of timer events result from attempting to layer soft-real-time tasks (Flash plugins) over a best-effort substrate (Firefox, and the Linux or Vista kernels).

The temporal behavior of a program which requires timeouts, watchdogs, periodic tasks, and the like can be viewed as a *cross-cutting concern* which affects areas of the system that do not correspond well to software module boundaries. The field of Aspect-Oriented Programming (AOP) has developed concepts and techniques for dealing with other such cross-cutting concerns, and it is an interesting open question as to whether such ideas can be applied to orthogonally specify the timer-related behavior of a software system. While a number of aspects of OS kernel design have been addressed by the AOP community including scheduling [1], we know of no work that has viewed the problem of layered timers from such a perspective.

5.3 A better notion of time

Analysis of call stacks in our traces shows that a significant number of periodic timers (and especially those with large human-specified periods) are intended to run background housekeeping activities such as page cleaning, or flushing the file-system journal. In these cases, the programmer probably meant:

> "Please wake up this thread at some convenient time
> in the next 10 minutes"

... rather than:

> "In 600.0 seconds time +/- 10ms, please execute the
> following function."

If the *precision* of a timeout is separately specified, the OS has the ability to batch timeout delivery, perhaps allowing the processor or disk to be placed in a power-saving mode. In reality, the programmer often has a mental utility function $u(t)$ for the value of running his code at some future time t. Various soft-real-time operating systems have explored task scheduling with similar abstractions [17, 32].

The recent changes to the Linux kernel interface described in Section 2.1 can be viewed as limited particular cases of this, designed to save power and/or CPU for timers whose expiry time does not need to be precise. However, this suggests that a timer subsystem would more generally benefit from a richer way of expressing an intended expiry time. Long-standing experience with declarative specification of results from relational databases shows that specifying the *minimum* necessary information allows a system maximum flexibility in generating suitable results—in this case, scheduling a timer.

Facilities including the Unix `cron` daemon already offer richer (and looser) specifications of periodic tasks, but we envision a timer subsystem offering still more expressive time values, such as:

> "Any time after 10 minutes," for a delay timer.
>
> "Every 5 minutes, on average over an hour," for a low-frequency periodic timer.
>
> "After we have exceeded 100 standard deviations above the mean round-trip time to this host," for a network-related timeout.

One concern with such flexible specifications is that the computational overhead of calculating a (nearly) optimal timer schedule should not outweigh any benefit gained from having a better schedule in the first place. However, a scheduling algorithm with such information can always fall back to simpler non-optimal schedules if system conditions require it, whereas the reverse is not true. Thus, we see this more as a continuous tradeoff between flexibility/optimality on the one hand, and overhead on the other. Different degrees of expressivity are clearly appropriate at different levels in the system, and we plan to investigate this tradeoff in more detail in our ongoing work.

5.4 Use-case-specific interfaces

Since most timer uses we observe can be fitted into a small number of well-defined use cases, we conjecture that we might improve the overall reliability of the system by replacing the general timeout mechanism with several abstractions, tailored to different usage scenarios. We explore the implementation advantages of this below; here we explore the design of such specialized interfaces.

The most obvious case for specialization is periodic tickers. The basic interface is:

> "Every time period of length t, invoke function f."

Periodic tickers requiring precision in timing would benefit from not having to reset themselves and correct for the time taken to do this in their calculation. Perhaps more importantly, periodic tasks requiring much less precise ticks could maintain average frequency while tolerating local variations in the interests of performance.

Timeouts provide a more interesting use-case, and indeed Win32 GUI developers already extensively use a programming idiom akin to that sometimes used for mutexes in C++: an auto object is declared in a procedure, its constructor installs a timeout, and its destructor cancels the timer. The net effect is to declare:

> "If this procedure has not returned in time t, invoke function e."

Specifying timeouts in this manner allows the timer implementation to identify the dependencies when nested timeouts are specified by code on the same thread. If the duration of an inner-level timeout exceeds an already-waiting timeout, the inner timeout may be ignored.

Watchdog timers are similar but subtly different to timeouts, although they offer similar scope for optimization. Their interface would be:

> "If this code path has not been executed within time t, invoke function f."

Finally, delay timers correspond most directly to the current timer subsystem API. They simply declare:

> "After time t, invoke function e."

These cases are derived from our observations about how a single set/cancel interface is used differently in practice by a variety of applications. However, the cases we have presented above point to a wider issue which has generally not been recognized by the designers of general-purpose OS kernels. The timer interface, when used in these ways, is telling the kernel which piece of code to run when. The kernel also has another subsystem dedicated to implementing this type of policy: the CPU scheduler.

5.5 Timers and scheduling

This has been a paper about timer subsystem design and usage and not CPU scheduling. However, it's clear that timers and scheduling are closely related—the CPU scheduler is in principle the system entity that decides what code to run when, whereas setting a timer implicitly requests that a piece of code run at a particular time in the future. Given this relationship, it is remarkable that both Linux and Vista have evolved with almost completely separate timer and scheduler subsystems, which interact almost solely through the operation of unblocking a thread.

At present, the timer system and the scheduler export separate interfaces, and indeed the timer system is practically the only way that an application can influence at a fine granularity how it is scheduled.

Our results in Section 4.3 suggest we can view the timer use of a soft-real-time application such as Skype or Firefox (with the Flash plugin) as an attempt to achieve a fine-grained scheduling behavior not provided by the CPU scheduler. More generally, the four identifiable use-cases in Section 4.1 suggest that an application-level interface to the CPU scheduler, rather than an explicit multiplexer of hardware timers, is what applications would find most useful.

Such a scheduler would differ from current designs in two respects: firstly, it would need to support an interface for applications to specify more complex requirements, such as those in Sections 5.3 and 5.4 above. We note that such application requirements need not violate any system-wide policies for allocating the CPU between tasks, but can provide for dispatching the application at the right time.

Secondly, in order to supplant the timer interface for applications, such a scheduler would need to dispatch the application in such a way as to run the right piece of code at the right time (as the timer interface purports to do), rather than simply resuming the process. This can be achieved either in the kernel or in user space libraries by the functionality of techniques such as Scheduler Activations [2] or the Psyche scheduler [23].

There are, of course, significant challenges to realizing such a design. In particular, the CPU scheduler must now deal with complex constraints (which can be thought of as short-term execution "plans", by analogy with database systems) from multiple applications as well as a system-wide CPU allocation policy, and these constraints change dynamically. Nevertheless, we feel this is a promising direction for future research and we are actively researching such a design.

Finally, we note that such insights are not entirely new in other, closely related fields. The soft real-time scheduling of multimedia operating systems such as Rialto [7] provides some of the functionality we are advocating here, though we propose a richer expression of application scheduling and callback requirements. Furthermore, the notion of eschewing timer events in favor of more sophisticated scheduling algorithms is commonplace in the real-time systems community. Hard real-time systems sometimes avoid timers completely except as a way of deterministically invoking the scheduler [12].

6. RELATED WORK

We are not aware of any previous work that takes an end-to-end view of timer use, or explores new interfaces for specifying timeouts. Surprisingly few researchers appear to have questioned the basic set/cancel interface for timers, focussing instead on point-solutions to problems such as timer resolution, timer overhead, and power consumption.

Higher-resolution timing subsystems have been investigated over a long time: UTIME [26] proposes a way to add sub-jiffy precision on the base of dynamic ticks to Linux in the context of *firm* real-time applications. In the same context, the Rialto operating system [7] proposes a very similar high-resolution timer subsystem. The HRT project [3], which eventually became the current implementation of high-resolution timers in Linux 2.6.16 [15], is a fork of the UTIME codebase. Stultz *et al.* [28] also proposed a new Linux timer subsystem with a view towards simplicity, correctness, clock source abstraction and high-resolution clocks, again on the basis of dynamic ticks. All five projects recognize the high overhead of timer management with high-resolution clock ticks.

There are several works on periodic polling of network interfaces for high-performance networks that inevitably deal with shortcomings of existing timer subsystems:

- *Soft timers* [4] is a facility to emulate a timer subsystem of microsecond precision without the processing overhead of hardware timer interrupts, by polling for timer expiry at convenient points in the execution of an operating system.
- Periodic timer interrupts are used in the Aurora ATM driver [25] to initiate polling for packet completions on Gigabit network interfaces, trading off between interrupt overhead and communication delay.

Brakmo and Peterson [9] report on shortcomings in TCP retransmit timeout estimates in the BSD4.4 implementation, to which they do not provide a final solution, but show that special cases can be improved.

The relationship between timer maintenance algorithms, time flow mechanisms used in discrete event simulations, and sorting techniques is explored by Timing Wheels [30]. This work presents an algorithm for time intervals of fixed size that takes O(1) time to maintain timers within that interval. Two extensions for dynamically-sized intervals of different granularity, based on hashing and hierarchical data structures, are presented as well.

7. CONCLUSION

We have presented a study of how timers are used by a series of workloads over both Linux and Vista, and outlined the challenges in collecting appropriate data, including correlating use of timers with applications, and tracing timer provenance through layers of multiplexing.

Our results show that timer usage is widespread in mainstream operating systems, and in particular is intensively used by graphical user applications and windowing systems. Moreover, the time values provided as arguments to the timer subsystems are in general fixed, round figures determined by humans, in sharp contrast to the timeouts set by protocols like TCP, which are carefully tuned based on online measurements. We have shown cases where the layering of timers loses information about the desired temporal behavior of the system as a whole.

We identified a number of clear and distinct usage patterns of timer subsystems. Based on these, we have proposed a higher-level interface to a timer subsystem that specifies more about the temporal behavior an application wants: what code needs to run when. We also argue for a more flexible expression of "time" that better reflects application requirements.

This finally has led us to reconsider the traditional separation between the timer subsystem and the CPU scheduler. Our ongoing research is investigating to what extent an application interface to the CPU scheduler (incorporating the use cases we have identified) obviates the need for a separate timer interface, and also the challenges in implementing such a CPU scheduler in a modern operating system.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their many insightful comments on how to improve this paper. We also acknowledge our shepherd, Steve Hand, for all his help, and finally the rest of the ETH Systems Group for their useful suggestions.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, and G. Muller. Evolving an OS kernel using temporal logic and aspect-oriented programming. In *ACP4IS '03: Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, March 2003.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

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

[4] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177. ACM, 2003.

[7] J. S. I. Barrerra, A. Forin, M. B. Jones, P. J. Leach, D. Rosu, and M.-C. Rosu. An overview of the Rialto real-time architecture. Technical Report MSR-TR-96-13, Microsoft Research (MSR), July 1996.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[9] L. S. Brakmo and L. L. Peterson. Performance problems in BSD4.4 TCP. *SIGCOMM Computer Communication Review*, 25(5):69–86, 1995.

[10] A. Chanda, A. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of EuroSys 2007*, Mar. 2007.

[11] M. Y. Chen, A. Accardi, E. Kıcıman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[12] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *SIGOPS Operating Systems Review*, 23(3):141–157, 1989.

[13] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference (LINUX-00)*, pages 63–72. The USENIX Association, Oct. 2000.

[14] R. Droms. RFC 2131: Dynamic host configuration protocol, Mar. 1997.

[15] T. Gleixner and D. Niehaus. Hrtimers and beyond: Transforming the Linux time subsystems. In *Proceedings of the Ottawa Linux Symposium (OLS'06)*, volume 1, pages 333–346, Ottawa, Ontario, Canada, July 2006.

[16] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the SIGCOMM'88 Symposium*, 18(4):314–329, Aug. 1988.

[17] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *Proceedings of IEEE RTSS*, pages 112–122, Dec. 1985.

[18] J. Lions. Lions' commentary on Unix 6th edition, May 1976. See source line 3845 ff.

[19] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 261–274, June 2001.

[20] A. Morton. zc and cyclesoak: Tools for accurately measuring system load and TCP efficiency. http://www.zipworld.com.au/~akpm/linux/#zc.

[21] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[22] I. Park and R. Buch. Event tracing: Improve debugging and performance tuning with ETW. *MSDN Magazine*, Apr. 2007. http://msdn.microsoft.com/msdnmag/issues/07/04/ETW/.

[23] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Multi-model parallel programming in Psyche. *SIGPLAN Notices*, 25(3):70–78, 1990.

[24] S. Siddah, V. Pallipadi, and A. van de Ven. Getting maximum mileage out of tickless. In *Proceedings of the Ottawa Linux Symposium (OLS'07)*, pages 201–208, Ottawa, Ontario, Canada, June 2007.

[25] J. Smith and C. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[26] 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 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, pages 112–120. IEEE, June 1998.

[27] R. W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, 1994.

[28] J. Stultz, N. Aravamudan, and D. Hart. We are not getting any younger: A new approach to time and timers. In *Proceedings of the Ottawa Linux Symposium (OLS'05)*, volume 1, pages 219–232, Ottawa, Ontario, Canada, July 2005.

[29] TwistedCore. http://twistedmatrix.com/trac/wiki/TwistedCore, August 2007.

[30] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*, pages 25–38. ACM Press, 1987.

[31] G. D. White, G. Nielsen, and S. M. Johnson. Timeout duration and the suppression of deviant behavior in children. *Journal of Applied Behavior Analysis*, 5(2):111–120, Summer 1972.

[32] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Time/utility function decomposition techniques for utility accrual scheduling algorithms in real-time distributed systems. *IEEE Transactions on Computers*, 54(9), September 2005.

[33] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Proceedings of the Ottawa Linux Symposium (OLS'03)*, 2003.