Fine Grained Kernel Logging with KLogger: Experience and Insights

Yoav Etsion, Dan Tsafrir, Scott Kirkpatrick, and Dror G. Feitelson

School of Computer Science and Engineering,

The Hebrew University, 91904 Jerusalem, Israel

E-mail: etsman@cs.huji.ac.il

Phone: +972 2 658 5571

Abstract

Understanding the detailed behavior of an operating system is crucial for making informed design decisions. But such an understanding is very hard to achieve, due to the increasing complexity of such systems and the fact that they are implemented and maintained by large and diverse groups of developers. Tools like Klogger — presented in this paper — can help by enabling fine-grained logging of system events and the sharing of a logging infrastructure between multiple developers and researchers, facilitating a methodology where design evaluation can be an integral part of kernel development. We demonstrate the need for such methodology by a host of case studies, using Klogger to better understand various subsystems in the Linux kernel, and pinpointing overheads and problems therein.

Keywords: Linux, kernel monitoring, overhead.

Introduction

In the late 1970s, UNIX version 6 consisted of ˜60,000 lines of code [16]. Today, version 2.6.9 of the Linux kernel consists of over 5,500,000 lines of code, and almost 15,000 source files. This is a great testimony to the complexity of modern operating systems.

Modern, general purpose operating systems need to manage a plethora of hardware devices: storage devices, networking, human interface devices, and the CPU itself. This is done using software layers such as device drivers, filesystems, and communications protocols. The software is designed and implemented by hundred of programmers writing co-dependant code. This is especially true for community-developed operating systems such as Linux and the BSD family. While such open-source approaches benefit from the talents and scrutiny of multiple avid developers, they may also lead to situations where different pieces of code clash, and do not interoperate correctly [2].

Adding to this problem is the immense power of modern CPUs. For one, the power of the hardware might mask performance problems. Secondly, as the achievable density of hardware components continues to grow exponentially, hardware vendors are increasingly employing parallelism, in the form of symmetric multi-processing (SMP) and even multi-core processors. Needless to say, supporting this increases system complexity.

The resulting software is too complex for a human programmer to contain, and might even display counter-intuitive behavior [14]. Analyzing system behavior based on measurements is often thwarted by measurement overheads that overshadow the effects being investigated, sometimes described as the *Heisenberg effect* for software [27]. All this has detrimental effects on the engineering of critical system components. For example, it is not uncommon that code is submitted to the Linux kernel, and sometimes even accepted, based on a subjective *"feels better"* argument [18].

This situation raises the need for better system analysis tools, that will aid developers and resarchers in obtaining a better understanding of system behavior. Given systems' complexity, one cannot expect an all ecompassing system analyzer, because that would require a full understanding of the operating system's code. A more promising approach is a framework allowing developers to build event loggers specific to the subsystem at hand. This framework should be integrated into the kernel development methodology, by designing a subsystem's event logger along with the subsystem itself. In fact, an event logger embodying the

subsystem's logic can also complement the subsystem's documentation. Such a framework may facilitate the creation of a collection of system loggers based on the experience of developers writing the code in the first place.

In this paper we introduce *Klogger*, a fine-grained, scalable, and highly flexible kernel logger. Klogger is designed as a post-mortem analysis tool, logging the configured kernel events with very low overhead. It is reliable and does not lose any events, which can be logged from any point in the running kernel. Logging is done into per-CPU buffers, making Klogger scalable, a required feature for the increasingly parallel modern processors. Klogger can be specialized for specific subsystems using an event configuration file, which leads to the generation of event-specific code at kernel compilation time. This structured specialization mechanism, called *Klogger schemata*, allows kernel developers to share their expertise and insights, thus allowing other researchers to analyze code without having to fully understand its intricacies.

Klogger currently supports the Linux kernel — both the 2.4.x and the 2.6.x kernel versions. Although a newer series exists, measuring the 2.4.x kernel series cannot be dismissed, as it is still the series of choice for many administrators, especially since the 2.6.x series is considered unstable even by some kernel developers [5, 32].

To demonstrate the power and flexibility of Klogger, we dedicate over half of this paper to describing several case studies in which Klogger uncovered bottlenecks or mis-features — including examples of what we have learned about the behavior of the Linux kernel using Klogger.

The rest of this paper is organized as follows: we start by reviewing other kernel loggers and analyzers. We then go on to describing Klogger's design principles and review several case studies demonstrating Klogger's flexibility.

Related Work

Klogger is a software tool used to log events from the operating system's kernel, with the developers defining the events at compilation time. This is not a novel approach, and there exist several tools which operate on the same principle. Unfortunately, these tools have various limitations, chief among which is high overhead that limits the granularity of events that can be investigated.

The simplest logging tool is *printk*, the kernel's console printing utility [4, 17], whose semantics are

identical to those of *C*'s standard *printf*. This tool incurs a substantial overhead for formatting, and is not reliable — it uses a cyclic buffer which is easily overrun, thus losing events, as it is read by an external, unsynchronized daemon.

The most effective Linux tool we have found is the *Linux Trace Toolkit* (LTT) [33]. LTT logs a set of some 45 predefined events, including interrupts, system calls, network packet arrivals, etc. This tool's effectiveness is witnessed by its relatively low overhead and a very useful visualization tool to help analyze the logged data. However, it is not flexible nor easily extensible to allow for specific instrumentation.

A more flexible approach is taken by *Kerninst* [29], and what seem to be its successors — *Dtrace* [8] on Sun's Solaris 10 operating system, and *Kprobes* [22] from IBM in Linux. These tools dynamically modify kernel code in order to instrument it: either by changing the opcode at the requested address to a jump instruction or by asserting the processor's debug registers, thus transferring control to the instrumentation code. After the data is logged, control returns to the original code. The ability to add events at runtime makes these tools more flexible than Klogger.

None of the above tools provide data about the overhead they incur per logging a single event (with the exception of *Kerninst*), which is the principal metric in evaluating a tool's granularity. We therefore measured them using the Klogger infrastructure and found that their overhead is typically much higher than that of Klogger. This measurement is described below (in the section dealing with Klogger's *stopwatch* capabilities), and is summarized in table 1.

TIPME [10] is a specialized tool aimed at studying system latencies, which logs system state into a memory resident buffer whenever the system's latencies were perceived as problematic. This tool partly inspired the design of Klogger, which also logs events into a special buffer. It is no longer supported, though.

The Windows family also has a kernel mechanism enabling logging some events, called *Windows Performance Monitors* [28], but very little is known about its implementation.

An alternative to logging all events is to use sampling [1]. This approach is used in *OProfile*, which is the underlying infrastructure for HP's *Prospect* tool . OProfile uses Intel's hardware performance counters [13] to generate traps every N occurances of some hardware event — be it clock cycles, cache misses, etc. The overhead includes a hardware trap and function call, and logging 10,000 events/second can lead to 3-10%

overall overhead (depending on which hardware counter is being used). Also, this tool is periodic, and thus bound to miss events whose granularity is finer than the sampling rate.

Yet another approach for investigating operating system events is to simulate the hardware. For example, *SimOS* [24] was effective in uncovering the coupling of the operating system and its underlying CPU [25], but is less effective when it comes to understanding the effects of specific workloads on the operating system per-se. Finally, architectures with programmable microcode have the option to modify the microcode itself to instrument and analyze the operating system, as has been done on the VAX [20]. In principle, this approach is also viable for Intel's PentiumIV processors, which internally map op-codes to μops using some firmware. The problem is that this firmware is one of Intel's best guarded secrets, and is not available for developers.

KLogger Design Principles

Klogger is a framework for logging important events to be analyzed offline. Events are logged into a memory buffer, which is dumped to disk by a special kernel thread whenever it's free space drops below some low-water mark.

The design of Klogger originated from the need for a tool that would enable kernel researchers and developers direct, unabridged, access to the "darkest" corners of the operating system kernel. None of the tools surveyed above provides the combination of qualities we required from a fine grained kernel logging tool. Thus, Klogger was designed with the following goals in mind:

A Tool for Researchers and Developers Klogger is targeted at researchers and developers, and not for production systems. This goal enforces us to maintain strict event ordering, so events are logged in the same order as executed by the hardware. Also, events must not get lost so logging must be reliable. These two features also make Klogger a very handy debug tool. On the other hand, this goal also allows for event logging code to incur some minimal overhead even when logging is disabled. An additional requirement was support for logging the hardware's performance counters. While such counters are now available on most platforms, we currently only support the Intel PentiumIV performance monitoring counters [13].

Low overhead When monitoring the behavior of any system, our goal is "to be but a mere fly on the wall". Thus overhead must be extremely low, so as not to perturb the system behavior. The overhead can be categorized into two orthogonal parts: *direct overhead* — the time needed to take the measurement, and *indirect overhead* — caused by cache and TLB lines evicted as a result of the logging. These issues are discussed below in the section dealing with Klogger's stopwatch capabilities.

Flexibility Klogger must be flexible, in that it can be used in any part of the kernel, log any event the researcher/developer can think of, and allow simplicity in adding new types of events. Also, it must allow for researchers to share methodologies: if one researcher comes up with a set of events that measure some subsystem, she should be able to easily share her test platform with other researchers, who are not familiar with the gritty implementation details of the subsystem at hand. This goal is important since it allows for Klogger users to easily incorporate the ideas and insights of others. Klogger's flexibility is further discussed in the section titled "Klogger Schemata" and demonstrated later on in several case studies.

Ease of Use Using Klogger should be intuitive. For this reason we have decided to use semantics similar to printing kernel data to the log, leaving the analysis of the results for later. These semantics, along with the strictly ordered, reliable logging makes Klogger a very useful debugging tool. Another aspect of this goal is that configuration parameters should be settable when the system is up, avoiding unnecessary reboots or recompilations. Klogger's programmer/user interface is further discussed below.

The designed goals are specified with no particular order. Even though we have found them to be conflicting at times, we believe we have managed to combine them with minimal tradeoffs. The rest of this section reviews the details of Klogger's interface and implementation.

KLogger: Programmer/User interface

This section will discuss the business end of Klogger – how to operate and configure this tool.

Klogger's operation philosophy is quite simple: when designing a measurement we first need to define what we want to log. In Klogger's lingo, this means defining an event and the data it holds. Second, we need to declare when we want this event logged. Third, we have to configure running parameters, the most important of which is the toggle switch — start and stop the measurement. The last step is analyzing the data,

the only part in which the user is on her own. Since analyzing the data is a task specific to the data gathered, the user needs to write a specific analyzing program to extract whatever information she chooses, be it averaging some value, or replaying a set of events to evaluate an alternate algorithm. To simplify analysis, Klogger's log is text based, and formatted as a Perl array of events, each being a Perl hash (actually, the log is dumped in its binary form, and later converted into its textual form using a special filter).

To simplify the description of the interface, we will go over the different components with a step by step example: defining an event that logs which process is scheduled to run. The event should be logged each time the process scheduler chooses a process, and should hold the *pid* of the selected process and the number of L2 cache misses processes experienced since the measurement started (granting a glimpse into the processes' cache behavior).

Event Configuration File

The event configuration file is located at the root of the kernel source tree. A kernel can have multiple configuration files — to allow for modular event schemata — all of which must be named with the same prefix, *.klogger.conf* (unlisted dot-files, following to the Linux convention for configuration files). The configuration file contains both the hardware performance counters definitions, and the event definitions.

Performance counter definitions are a binding between a virtual counter number and an event type. The number of counters is limited only by the underlying hardware, which has a limited number of registers. Sometimes certain events can only be counted using a specific subset of those registers, further limiting the performance counters variety. In our example we set virtual hardware counter 0 to count L2 cache misses (counter names are implementation dependant):

```
arch PentiumIV {
    counter0 l2_cache_misses
}
```

Accessing a hardware counter is described below.

Event definitions are *C*-like structure entities, declaring the event's name and the data fields it contains. The event used in our example is

```
event SCHEDIN {
    int pid
    ulonglong L2_cache_misses
}
```

This event will be called *SCHEDIN*, and will have three fields — the two specified, and a generic header which contains the event type, its serial number in the log, and a timestamp indicating when the event occurred. The timestamp is taken from the underlying hardware's cycle counter, which produces the best possible timing resolution. This event will appear in the log file as the following Perl hash:

```
{
    header => {
        "type"      => "SCHEDIN",
        "serial"    => "119",
        "timestamp" => "103207175760",
    },
    "pid"              => "1073",
    "L2_cache_misses"  => "35678014",
},
```

A more detailed description of the configuration file is beyond the scope of this paper.

Event Logging

Logging events inside the kernel code is similar to using the kernel's *printk* function. Klogger calls are made using a special *C* macro called *klogger*, which is mapped at preprocessing time to an *inlined* logging function specific to the event. This optimization saves the function call overhead, as the klogger logging code simply stores the logged data on the log buffer.

The syntax of the logging call is:

```
klogger(EVENT, field1, field2, ...);
```

where the arguments are listed in the same order as they are declared in the event definition. Klogger uses *C*'s standard type checks. In our scheduler example, the logging command would be:

```
klogger(SCHEDIN, task->pid,
        klogger_get_l2_cache_misses());
```

with the last argument being a specially auto-generated inline function that reads the appropriate counter.

At times, even though the kernel is capable of logging a variety of events, we want to disable some so only a subset of the events actually get logged. This is done by using the Linux *sysctl* interface, or its equivalent */proc* filesystem counterpart. Each event is allocated a specific *sysctl* value, and a corresponding file on the */proc* filesystem. Writing a value of 0 or 1 to this file disables or enables that event, respectively.

Configuration Parameters

In addition to the per-event controls, Klogger also has a general on/off switch. By writing 1 into the */proc/sys/klogger/enable* file, the user switches on logging. Writing 0 into that file turns logging off. The file can also be read to determine whether the system is currently logging. Accessing this on/off switch using the filesystem greatly simplifies Klogger usage, as it enables users to write shell scripts executing specific scenarios to be logged. It also allows a running program to turn on logging when a certain phase of the computation is reached.

Another important configuration parameter is the buffer size, set by default to 4MB. However, as the periodic flushing of the buffer to disk obviously perturbs the system, a bigger buffer is needed in scenarios where a measurement might take longer to run and the user does not want it disturbed.

The last parameter worth mentioning is the low-water mark. This parameter determines when the buffer will be flushed to disk, and its units are percents of the full buffer. Klogger's logging buffer acts as an asymmetric double buffer, where the part above the low-water mark is the main buffer, and the part below the low-water mark is the reserve buffer, that is only used when flushing the buffer. This mechanism is further explained in the following section. By default, the buffer is flushed when its free space drops below 10%. In some scenarios — when logging filesystem accesses for example, the flushing action itself generates events — the threshold should be increased, to avoid overflowing the buffer. If an overflow does occur the kernel simply starts skipping event serial numbers, until space is available. This way the log's integrity can

be easily verified — allowing logged events to be skipped if no memory resources are available, informing the user of such occurrences.

Internal Benchmarking Mechanism

The final part of Klogger's interface is its internal benchmarking mechanism. When designing a benchmark, one needs to pay attention to the overhead incurred by the logging itself, in order to evaluate the quality of the data collected. For each event, Klogger measures the time consumed by its logging process. This data is exported using the */proc* filesystem.

Finishing with our example, we should mention that logging our *SCHEDIN* event takes ˜200 cycles on our 2.8GHz PentiumIV machine — or **˜71 nanoseconds**.

Klogger Implementation

In this section we discuss the details of Klogger's implementation and how its design principles — mainly the low overhead and flexibility — were achieved.

Per-CPU Buffers

As noted previously, Klogger's buffer operates as an asymmetric double buffer, with the low-water mark separating the main buffer from the reserve, flush time, buffer.

Klogger employs per-CPU, logically contiguous, memory locked buffers. In this manner allocating buffer space need not involve any inter-CPU locks, but only care for local CPU synchronization (as opposed to the physical memory buffer used in [9]). On a single CPU, race conditions can only occur between system context and interrupt context, so blocking interrupts is the only synchronization construct required. In fact, since the buffer is only written linearly, maintaining a *current position* pointer to the first free byte in the buffer is all the accounting needed, and safely allocating *event_size* bytes on the buffer only requires the following operations:

1. block local interrupts
2. event_ptr = next_free_byte_ptr

3. next_free_byte_ptr += event_size

4. unblock local interrupts

Interrupt blocking is required to prevent the same space allocated to several events, since the *next_free_byte_ptr* pointer is incremented on every event allocation. Furthermore, we want to prevent the possibility that the buffer will be flushed between the event allocation and the actual event logging. As flushing requires the kernel to context switch into the kernel thread in charge of flushing the specific per-CPU buffer (described below), disabling kernel preemption during the logging operation assures reliability (in fact, kernel preemption is possible only if explicitly enabled upon kernel compilation). Needless to say this simple synchronization hardly interferes with the kernel's normal operation, as it only involves intra-CPU operations, thus granting Klogger its scalability, allowing it to be efficiently used in SMP environments.

Logging buffers are written-to sequentially, and only read-from at flush time. With these memory semantics, it is obvious that caching does not improve performance, but quite the contrary: it can only pollute the memory caches. We therefore set the buffers' cache policy to *Write-Combining* (WC). This semantic, originally introduced by Intel with its *PentiumPro* processor [13], is intended for memory that is sequentially written-to and is rarely read-from, such as frame buffers. WC does not cache data on reads, and accumulates adjacent writes on a CPU internal buffer, only to write them in one bus burst.

Per-CPU Threads

During the boot process, Klogger spawns per-CPU kernel threads, that are in charge of flushing the buffers when the low-water mark is reached. Although the logging operation should not disturb the logged system, flushing the buffer to disk obviously does. To minimize the disturbance Klogger threads run at the highest priority under the real time SCHED_FIFO scheduler class. This class, mandated by Posix, has precedence over all other scheduling classes, meaning no process is able to prevent the Klogger threads from running.

Each thread dumps the per-CPU buffer to a per-CPU file. The separate files can later be interleved using timestamps in the events' headers, as Linux synchronizes the per-CPU cycle counters on SMP machines [4, 17].

Flushing the buffer might cause additional events to be logged, so the buffer should be flushed before it is totally full. Klogger's low-water paramter mentioned above, is there for just that: invoking the flush

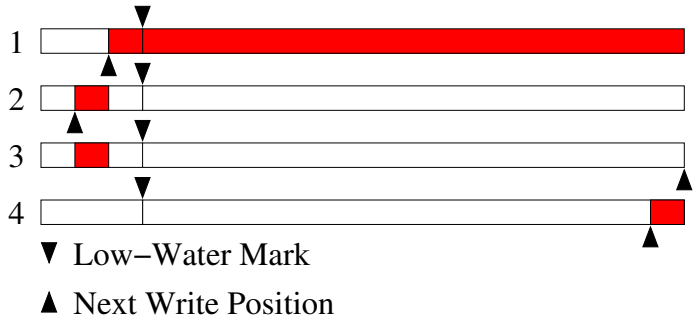

Figure 1: *The four steps of the flush operation: (1) The log buffer reaches the low-water mark and wakes up the dump thread. (2) Thread writes the data between the beginning of the buffer and the current position, possibly causing new events to be logged to the reserve part. (3) Atomically resetting the buffer's current position (with interrupts disabled). (4) Events from the reserve part are flushed to disk, possibly causing new events to be logged at the beginning of the buffer.*

before the buffer is filled, while utilizing its bulk to minimize flushes. As such, flushing the buffer is a four step process, which carefully flushes both the main part of the buffer and the reserve part (everything under the low-water mark). The steps are described in Figure 1.

To prevent logged data from being tainted by Klogger-induced events, buffer flushes are marked in the log by special events: *DUMP_BEGIN* and *DUMP_FINISH*. The presence of these two events in the log allows for cleanning the data from artifacts introduced by the logging function itself, further deminishing the Heisenberg effect.

When Klogger is disabled, the Klogger threads are awakened in order to empty all CPU buffers, and only then is Klogger ready for another logging session.

Code Generation

Klogger generates specific encoding and decoding code for each user defined event, as two complementing inlined *C* functions.

The decision to generate specific code for each event, rather than use generic code, is motivated by the desire to reduce the overhead as much as possible. An important part of Klogger is it's simple, yet powerful code generator. The generator produces specially crafted code for each event that simply allocates space on the CPU's buffer and copies the logged data field by field. The code avoids using any branches or extra memory which might cause cache misses, in order to reduce the uncertainty induced by the logging action as

much as possible. It is optimized for the common code path: successfully allocating space and logging the data, incurring only one forward branch overhead when the buffer is full. The resulting overhead is indeed minimal, as reported in the *Stopwatch* schema case-study.

Neither the code generation nor executing the event specific code requires intervention from the user — generating the code is an implicit part of the compilation process, and event logging is done using the generic Klogger *C* macro which is directed to the specific encoder by the *C* preprocessor.

Extent of Changes to the Linux Kernel

Knowing the complexity of the Linux kernel, and the rate of code evolution therein, we have tried to make Klogger's code self contained in its own files, non-intrusive to the kernel sources.

The full Klogger patch consists of about 4600 lines of code, of which, under 40 lines modify kernel sources, and only 13 modify kernel Makefiles! The rest of the patch consists of Klogger's own files. This fact makes Klogger highly portable between kernel versions — the same patch can apply to several minor kernel revisions.

Moreover, Klogger only uses a minimal set of kernel constructs: kernel thread creation, memory allocation, atomic bit operations, and just a few others. As such, porting it to other operating systems should be a feasible task.

Klogger Schemata

Klogger's schemata are its most powerful mode of operation. A schema is simply a set of complementary events, that provide comprehensive coverage of a certain subsystem or issue. For example, if Klogger is set up to log all kernel interrupts, we say it is using the Interrupt Logging Schema (our own Interrupt Logging Schema is described in section). Such schemata turn Klogger into a flexible framework enabling easy instrumentation of kernel subsystems, but more importantly, they provide a platform with which the research community can discuss and standardize the evaluation of these subsystems. This modular design enables the evaluation of separate subsystems individually, but also as a whole.

In practice, a Klogger schema is composed of a set of Klogger configuration files, and a kernel patch incorporating the necessary Klogger calls. Such a kernel patch is considered a light patch, as it just places

Klogger calls in strategic locations. This combination gives Klogger schemata the power of simplicity: first, it is very easy to create new schemata, assuming one knows her way around the kernel well enough to place the Klogger calls. Second, it is also very simple to apply an existing schema — one simply copies the Klogger configuration files associated with the schema, and applies the kernel patch.

While Klogger simplifies the process of evaluating kernel subsystems, creating a new schema requires a good understanding of the subsystem at hand. Our vision is to collect a host of schemata, created by kernel researchers and developers, incorprating their knowledge and insights. In particular, developers of new kernel facilities just need to write a schema able to log and evaluate their work. We believe such a collection can be a valuable asset for the operating system research community.

The following sections will describe some case studies utilizing a few of the basic schemata we desgined, and show some interesting findings and insights we have gathered when using Klogger.

Testbed

Our case studies demonstrating Klogger's abilities were conducted on klogger-enhanced 2.6.9 and 2.4.29 Linux kernels, representing the 2.6 and 2.4 kernel series, respectively. Klogger was set to use a 128MB memory buffer, to avoid buffer flushing during the measurements.

Our default hardware was a 2.8GHz PentiumIV machine, equipped with 512KB L2 cache, 16KB L1 data cache, 12Kμops L1 instruction cache, and 512MB RAM. Other hardware used is specified when relevant.

Case Study: Stopwatch Schema

The *Stopwatch* schema uses two event types: *START* and *STOP*. As the name suggests, it is used to measure the time it takes to perform an action, simply by locating the two events before and after the action takes place. In fact, when used in conjunction with the hardware performance counters it can measure almost any type of system metric: cache misses, branch mis-prediction, and instructions per cycle (IPC), just to name a few. Creating this schema required no more the a few minutes.

Table 1: The mean overhead ± standard deviation incurred by different logging facilities, measured using the *Stopwatch* schema. Direct overheads are shown in cycles, after subtracting the Stopwatch events' overhead.

Tool	Direct Overhead	L1 Cache Misses
KLogger	334±555	6.64±3.91
LTT	1873±1424	69.31±27.43
printk	4251±86	227.46±3.45
H/W Trap	485±265	*N/A*

Measuring Kernel Loggers

A good demonstration of Klogger's flexibility is its ability to measure the overhead incurred by other logging tools. The general overhead of a tool can be defined as how much it interferes with the test subject. We have used three interference metrics: *direct overhead*, the number of computing cycles consumed by the logging action, *L1 cache misses*, and *L2 cache misses* which indicates the number of cache lines evicted in the respective cache due to the logging action — a well known cause of uncertainty in fine grained computation, and in operating systems in general [30].

The logging tools we have measured are *printk* [17], whose semantics Klogger uses, *Linux Trace Toolkit (LTT)* [33], a well known logging tool in the Linux community, and Klogger itself. In order to create a meaningful measurement, we needed the logging mechanisms to log the same information — so we simply implemented a subset of LTT's events as a Klogger schema. Another promising tool is Sun's *DTrace* [8], which is an intergral part of the new Solaris 10 operating system. At the time of writing, however, we did not have access to its source code. Instead, we estimated its direct overhead by measuring the number of cycles consumed by a hardware trap (which is the logging method used in the *x86* version of Solaris). A hardware trap is also at the core of the *Kprobes* tool.

Table 1 shows the results of one of the most common and simple events — checking if there is any delayed work pending in the kernel (softirq). This event is logged at a rate of 1000Hz in the 2.6.x Linux kernel series, each time saving just two integers to the log. It is clear that Klogger incurs much less overhead than the other tools: by a factor of 5 less than LTT, and more than an order of magnitude for printk. The difference between indirect overheads is even greater (we only show L1 misses, as L2 misses were negligible for all tools). As for Dtrace, while Klogger incurs less overhead than a single hardware trap — the basic building block of the DTrace tool on the x86 architecture — we only see a 45% difference in the direct

overhead. However, as DTrace is based on a virtualized environment, it can be assumed that its direct overhead is actually considerably greater.

The fact that Klogger can be used to measure the performance and overheads of other logging tools is a tribute to its efficiency and versatility.

Case Study: Locking Schema

Modern operating systems employ fine grained mutual exclusion mechanisms in order to avoid inter-CPU race conditions on SMPs [3, 26]. Klogger's *locking schema* is intended to explore the overheads of using inter-CPU locks.

Fine grained mutual exclusion in Linux is done through two basic busy-wait locks: *spinlock* and *rwlock* [4, 17]. The first is the simplest form busy-wait mutual exclusion, where only one CPU is allowed inside the critical section at any given time. The second lock separates code that does not modify the critical resource — a *reader* — from code that modifies that resource — a *writer*, allowing multiple readers to access the resource simultaneously, while writers are granted exclusive access.

The goal of the locking schema is to measure lock contention, and identify bottlenecks and scalability issues in the kernel. The schema tracks the locks by the locking variable's memory address, and is composed of 5 events. Two are initialization events (*RWINIT*/*SPININIT*) which are logged whenever Klogger first encounters a lock — these events log the lock's address and name (through *C* macro expansion). The three other events — *READ*, *WRITE*, and *SPIN* — are logged whenever a lock is acquired. Each log entry logs the lock's address, the time it was acquired and the time it was released, thus allowing to compute the number of cycles spent spinning on the lock, while reducing overhead by logging a single event for every lock acquisition rather than two (lock + release). This schema is the most intrusive as it wraps the kernel's inlined lock functions with macros to allow for accounting. Still, its overhead is only ˜10% of the cycles required to acquire a free lock (let alone a busy one).

Overhead of Locking

How many cycles are spent by the kernel spinning on locks? Very little data is known on the matter: Bryant and Hawkes [7] wrote a specialized tool to measure lock contention in the Linux kernel, which they used

to analyze filesystem performance [6], but it is no longer maintained. Kravetz and Franke [15] focused on contention in the 2.4.x kernel CPU scheduler, which has since been completely rewritten. A more general approach was taken by Mellor-Crummey and Scott [19], who measured the overhead of acquiring a lock, running without an operating system, while Unrau et al. [31] showed the same overhead for the experimental *Hurricane* operating system — both using hardware that is now considered antiquated, so their results are irrelevant for evaluating the overall overhead of locking on common workloads, hardware, and operating systems. Such an evaluation is becoming important with the increasing popularity of SMP (and the emerging multi-core) architectures in servers and on the desktop.

Locking is most pronounced with applications that access shared resources, such as the virtual filesystem (VFS) and network, and applications that spawn many processes. In order to identify contended locks, we chose a few applications that stress these subsystems, using varying degrees of parallelization.

- **Make**, running a parallel compilation of the Linux kernel. This application is intended to uncover bottlenecks in the VFS subsystem. In order to isolate the core VFS subsystem from the hardware, compilations were performed both on memory resident and disk based filesystems.
- **Netperf**, a network performance evaluation tool. We measured the server side, with multiple clients sending communications using the message sizes in Netperf's standard round-robin TCP benchmark — 1:1, 64:64, 100:200, and 128:8192, where the first number is the size of the message sent by the client, and the second is the size of the reply. Each connecting client causes the creation of a corresponding Netperf process on the server machine.
- **Apache**, the popular web server was used to stress both the network and the filesystem. Apache was using the default configuration, serving Linux kernel source files from a RAM based filesystem. To simulate dynamic content generation (a common web server configuration), the files are filtered throught a Perl CGI script that adds line numbers to the source files. Stressing was done using the Apache project's own flood tool. Its performance peaked at 117Req/s.

In this case study we used the largest SMP available to us: a 4-way PentiumIII Xeon processors (512KB L2 cache), equipped with 2GB of RAM. It's network interface card (NIC) is a 100Mb/s Ethernet card. The stressing clients are a cluster of 2-way PentiumIV 3.06GHz machines (512KB L2 cache, 4GB RAM),

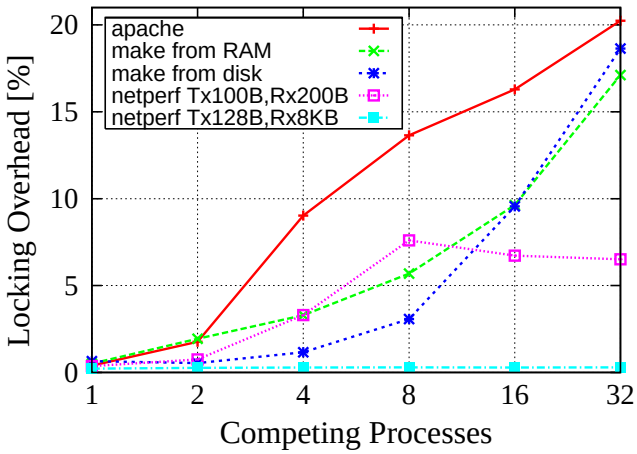

Figure 2: *Percentage of cycles spend spinning on locks for each of the test applications.*

equipped with 1Gb/s Ethernet cards. Klogger was set with a 128MB buffer for each of the server's CPUs. In order verify the results from the somewhat aging hardware are indeed relevant, we repeated all tests on the same hardware running in 2-way SMP mode and compared the obtained results with those running on the modern 2-way SMP clients. The similarity of these results indicate that although the processors are older, the SMP behavior of the systems has not changed. For lack of space, we only show the results for the 4-way SMP hardware.

Tests consisted of running each application with different levels of paralellism — 1,2,4,8,16, and 32 concurrent processes: when N was the degree of parallelism, Make was run with the *-jN* flag spawning N parallel jobs, while Apache and Netperf simply served N clients. During test execution Klogger logged all locking events within a period of 30 seconds. The reason for this methodology is that the kernel uses locks very frequently, generating a huge amounts of data. The 30 seconds period was set so Klogger could maximize its buffer utilization, while avoiding flushing it and interfering with the measurement.

Using the logged data, we aggregated the total number of cycles spent on locking in the kernel. The results are shown in Figure 2 (the actual overhead is bigger, since we only accumulate the time spent spinning, neglecting the overhead of calling the lock in the first place).

At the highest level of parallelism, running Apache has the CPUs spend over 20% of their cycles waiting for locks, and both measurements of Make exceed 15% overhead. Netperf however, suffers just over 6% overhead simply because the 100Mb/s network link was saturated. If we focus on the point of full utlization,

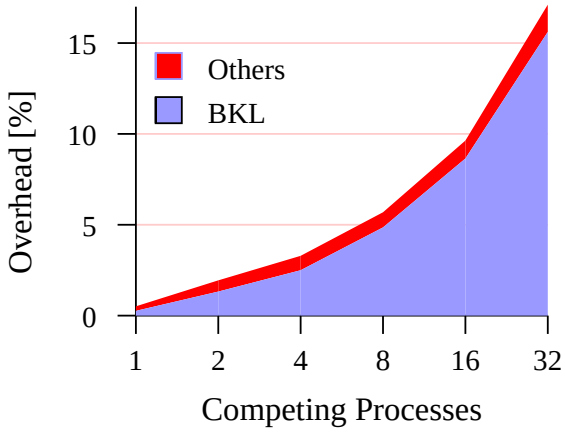

Figure 3: *Portion of the cycles spend on BKL and other locks, for ramdisk-based Make.*

which is at 4 competing processes for our 4-way SMP, we see that Apache loses ˜9% to spinning. This is a substantial amount of cycles that the CPUs, well, just wait!

The case of the Make benchmarks is especially interesting. When using a memory based filesystem vs. a disk based one, we would expect better performance from the memory based filesystem, as it does not involve accessing the slower hard disk media. But when using 4 processes, the results for both mediums were roughly the same. The answer lies in the locking overhead: while the ramdisk based Make loses just over 3% to spinning, the disk based one loses just over 1%. It appears time spent by processes waiting for disk data actually eases the load on the filesystem locks, thus compensating for the longer latencies.

The next step was to identify the bottlenecks: which locks are most contended? It seems the cause of this behavior in all but the Netperf example is just 1 lock — Linux's *Big Kernel Lock* (BKL).

The BKL is a relic from the early days of Linux's SMP support. When SMP support was first introduced to the kernel, one processor was allowed to run kernel code at any given time. The BKL was introduced somewhere between the 2.0.x and 2.2.x kernel versions as a hybrid solution that will ease the transition from this earlier monolithic SMP support, to the modern, fine grained support. It's purpose was to serve as a wildcard lock for subsystems not yet modified for fine-grained locking. The BKL has been deemed a deprecated feature for quite some time, and developers are instructed not to use it in new code. It is still extensively used, however, in filesystem code, and in quite a few device drivers.

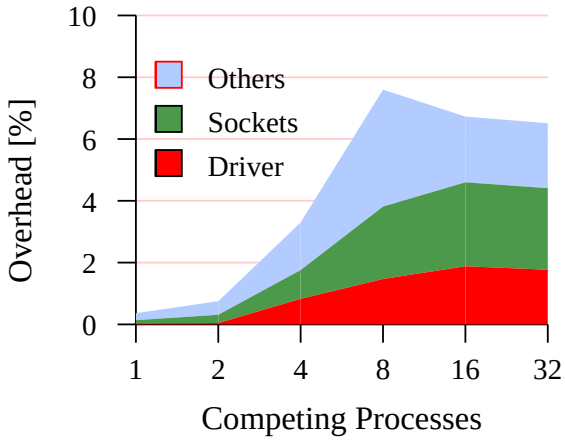

Figure 4: *Overheads of different lock types for the Netperf benchmark, using the 100:200 message sizes (client sends 100B, receiving 200B).*

Figure 3 shows the portion of the BKL in the overall lock overhead for the ramdisk based Make benchmark. Results for the disk-based version and Apache are similar. Obviously BKL accounts for the lion's share of the overhead, with all other locks taking no more than 2% of the overall cycles, and only roughly 0.5% in the common case. In particular, we found that the memory-based Make accesses BKL twice as often as the disk-based one.

The picture is completely different for the Netperf benchmark (Figure 4). BKL is completely missing from this picture, as both the networking and scheduling subsystems were completely rewritten since the introduction of BKL, and have taken it out of use. Instead, locking overhead is shared by the device driver lock, socket locks, and all other locks. The device driver lock protects the driver's private settings and is locked whenever a packet is transmitted or received and when driver settings change — even when the device's LED blinks! basically, this lock is held almost every time the device driver code is executed. In fact, it is locked more times than any other lock in the system by a factor of almost 3. The socket locks refer to *all* the sockets in the system, meaning at least the number of running Netperf processes: each Netperf process owns one socket. This figure is a rough estimate of the aggregate locking overhead caused by the networking subsystem. Both the device driver lock and the socket locks indicate the saturation of the networking link when running somewhere between 8-16 competing processes. All other locks in the system are responsible

for ˜33% of all the locking activity, peaking when running 8 competing processes. The majority of those cycles are spent on various process wait queues, probably related to the networking subsystem. We did not, however, find any group of locks causing the 8 process peak.

In conclusion, our measurements demonstrate the continuing liabilities caused by BKL even in the recent 2.6.9 kernel, and the harmful effects of device drivers with a questionable design. This is just a simple analysis of critical fine-grained locking mechanisms in the Linux kernel, made possible by Klogger's low overhead. The fact that we immediately came by such bottlenecks only strengthens the assumption that many more of these performance problems are found in the kernel, but we simply lack the tools and the methodology to identify them.

Case Study: Scheduler Schema

The *scheduler schema* consists of 8 basic events which allow for an accurate replay of process CPU consumption. Essential information about each event is also logged. The events are:

1. *TRY_TO_WAKEUP* — some process has been awakened.
2. *REMOVE_FROM_RUNQ* — a process has been removed from the run queue.
3. *ADD_TO_RUNQ* — a process has been added to the run queue.
4. *SCHEDOUT* — the running process has been scheduled off a CPU.
5. *SCHEDIN* — a process has been scheduled to run.
6. *FORK* — a new process has been forked.
7. *EXEC* — the *exec* system call was called
8. *EXIT* — process termination.

Using Klogger, creating these events is technically very easy. However, designing this schema and the data it logs requires in-depth knowledge about the design and behavior of the Linux CPU scheduler. An example of this is the *SCHEDIN* event: in Linux, a process' return path from a kernel after it has been scheduled to run for the first time (right after a *fork*) is different from that taken in the successing times it is scheduled to run. This fact makes the logging of *SCHEDIN* events somewhat delicate, for misplacing its call in the kernel code might cause the first run of every newly created process to be missed. This is a good

demonstration why a logging tool cannot be built to encompass *all* kernel subsystems, without help from researchers in various fields, and the importance of a rich collection of Klogger schemata to the research community.

Evaluating the Scheduler's Maximal Time Quantum

Klogger's scheduling schema can be used to empirically evaluate aspects of the Linux scheduler's design. The maximal CPU timeslice is an example for a kernel parameter that has changed several times in the past few years. It was a default 200ms in the 2.2.x kernels. The 2.4.x kernels set it to a default 60ms, but it could be changed in the 10–110ms range based on the process's dynamic priority. Today, the 2.6.x kernels set it to a value in the range of 5–800ms based on *nice* (the static priority),with a 100ms default when nice is 0, which it nearly always is. When searching the Linux kernel mailing list we failed to find any real reasoning behind these decisions.

An interesting question is whether these settings matter at all. We refer to an *effective quantum* as the time that passed from the moment the process was chosen to run on a processor, until the moment the processor was given to another process (either volutarily or as a result of preemption). In this case study, we wish to determine whether the effective quanta of various common workloads correspond to the maximum quantum length. (in a previous paper [11], we showed the effects of the clock resolution on the average effective quanta duration)

Using Klogger's scheduling schema, determining the fitness of the maximum quanta is very simple. The applications were chosen as representatives of a few common workloads:

- **Multimedia** — Playing a 45 second MPEG2 clip using the multithreaded *Xine* and the single threaded *MPlayer* movie players. Both players are popular in the Linux community, with xine's library being the display engine behind many other movie players.
- **Network** — Downloading a ˜30MB kernel image using the *wget* network downloader.
- **Disk Utilities** — Copying a 100MB file, and using *find* to search for a filename pattern on a subtree of the */usr* filesystem.
- **Computation+Disk** — Kernel compilation.

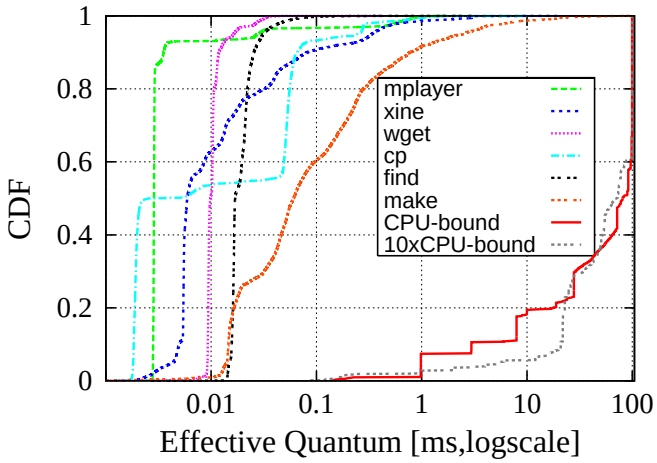

Figure 5: *Cumulative distribution function (CDF) of the effective quanta for the different workloads. Quantum length is 100ms. X axis is logarithmic.*

- **Pure Computation** — A synthetic CPU-bound program, continuously adding integers and never yielding the CPU voluntarily, running for 60 seconds.

Measurements were run with the default *nice* value, meaning a 100ms maximum time quantum on the 2.6.9 Linux kernel. We ran each application on a dedicated machine. The results are shown in Figure 5.

Let us first discuss the synthetic CPU-bound application: even this synthetic workload only reaches the maximum quantum in ˜40% of its runs, with a similar percentage not even reaching half the maximal quantum. Thus ˜60 % of the quanta were shortened by some system daemon waking up — the only background load in the system. These interruptions hardly consumed any CPU: only 0.00675% of the total run! They occured at an average rate of 5 times per second. With the maximal quantum set at 100ms, at most 50% of the quanta should have been affected, contrary to the results displayed in figure 5 which show that 60% were affected. The explanation is simple: an interrupted quantum is split into at least two effective quanta (a quantum can be broken down more than once), so the effect of the noise generated by system applications is actually amplified.

As for the other workloads, it is clear that the maximum quantum is almost a theoretical bound that is never reached: ˜90% of the effective quanta of all applications but Make and the CPU-bound are shorter then 100μs — a thousandth of the maximum quantum. The kernel Make is an exception, with its 90th percentile lying at 1ms (this is still a negligible 1% of the maximum quantum). In fact, if not for the logarithmic scaling

of the X axis we would not have been able to see any differences.

Our conclusion is that although required to prevent starvation, the actual length of the time quantum has little importance in modern systems. The only workload affected by it is CPU-bound. It would need to be shortened by more than 100 to affect other application types (regardless of what the effect would actually be), but as Linux currently uses a 1000Hz clock (on the *x86* architecture) it cannot support a sub-millisecond quantum. Lengthening the maximum time quantum on CPU servers in an attempt to reduce the context switch overhead (measured using Klogger to be $3608_{\pm 1630}$ cycles and $140_{\pm 38}$ L1 misses) is also futile in light of the scheduling noise generated by system daemons. This is an example of how a consistent use of logging tools such as Klogger by kernel devlopers can help make more informed decisions about parameter settings, adapting them to common workloads.

When 8 Competitors Are (Slightly) Better Than 1

During our work we needed to evaluate the effects of multiprogramming on the overall throughput of the computation. Our testbed was the Linux 2.4.29 kernel, and the benchmark we used was a program sorting an integer array whose size is one half the L2 cache. Our throughput metric is simple: how many times was the array sorted during a specified time frame? (the array was reinitialized to the same random values after each sort). We expected that this CPU-bound benchmark would achieve lower aggregate throughput if we ran several competing copies of it, since that would require the operating system to spend CPU time on context switching (with its corresponding cache pollution).

Our results, however, showed that throughput improved slightly with more processors, and peaked at 8 — a ˜0.3% improvement. In fact, this slight difference almost tempted to dismiss it, but since it was consistent we decided to check whether Klogger can help explain this discrepancy. Using the scheduler schema, we measured the CPU scheduling overhead, only to find it has a U shape (Figure 6). In particular, the total time spent on context switches (accumulating the time between all *SCHEDOUT* events and their immediately following *SCHEDIN* events) was much greater for the single process case than for the 8 process case: 30.20ms vs. 13.24ms respectively.

Unearthing the reason for this required a careful examination of the kernel's scheduling code. The 2.4.x scheduler linearly iterates over all the runnable processes to choose the one with the highest priority.

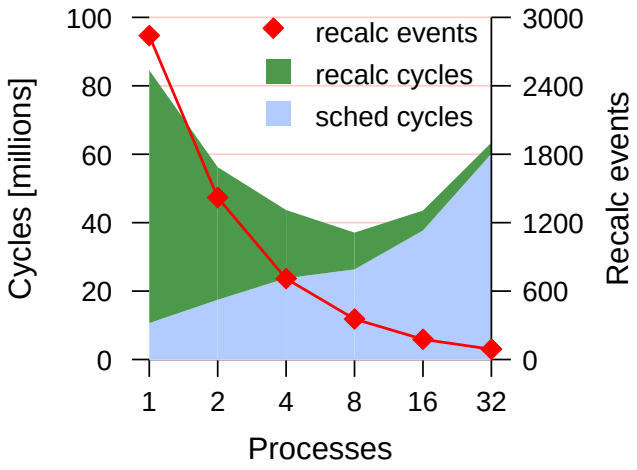

Figure 6: *Analysis of the scheduling overhead into its components: choosing a new process to run (bottom) and recalculating priorities. Also shown are the number of recalculations for each run.*

If no runnable process exists, the scheduler iterates over all existing processes, recalculating their CPU timeslice [4]. When running the benchmark with a single process this recalculation took place at almost every scheduling point. This is inefficient, as it considers dozens of system daemons which are dormant most of the time. With more user processes the frequency of these recalculations was decreased, saving much overhead. The time to recalculate was slightly longer, as there were more processes in the system, and the time to select one for execution was also longer, but these only grew enough to dominate the overhead at more than 8 processes — leading to 8 being the sweet spot.

Even though both the recalculation and the process selection loops were eliminated from the kernel as part of a complete scheduler redesign [17], between 2.4.x and 2.6.x versions, this case study still serves as a good example of how Klogger was used to understand unpredictable results which were initially attributed to common system noise. Using Klogger we were able to correctly link those results to a specific design issue.

Case Study: Interrupt Schema

Klogger's *interrupt schema* measures the start and finish of all interrupts in the system, including IRQs, traps, and exceptions, as well as L1 and L2 cache misses caused by the operating system code.

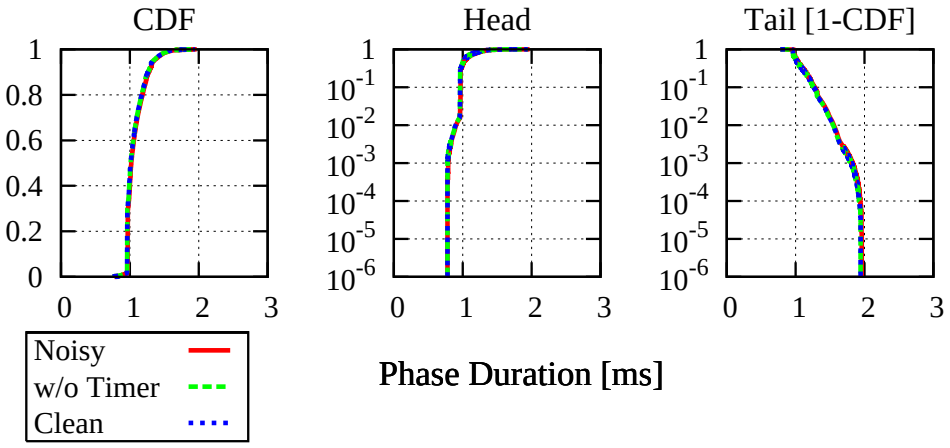

Figure 7: *CDF of a 1ms loop running with the FIFO scheduler (left), zooming in on the the head (center) and the tail (right). Head and tail are shown using a log scaled Y axis. The graphs illustrate the raw intervals ("Noisy"), the intervals after removing the direct overhead of the timer ticks ("w/o Timer"), and the intervals with all interrupts' direct overhead removed ("Clean").*

Operating System Noise

Noise caused by the operating system is becoming a growing concern. Interrupts, scheduling, TLB and cache contention are all causes for computational uncertainty, affecting multimedia and HPC applications [12, 23, 30].

In order to characterize interrupt noise we designed a synthetic application, based on a calibrated loop taking 1ms on average. The only memory activity is reading the counting index from the stack, incrementing it, and writing it back to the stack. This loop is repeated 1,000,000 times, keeping track of the number of CPU cycles consumed by each repetition. We ran the application on a klogger-enabled 2.6.9 Linux kernel under the Posix FIFO scheduler, so the only operating system noise that can disrupt the application is hardware interrupts.

Figure 7 shows a CDF of the repetition's times, zooming in on the head and tail. The figure shows that over 40% of the repetitions exceed 1ms, and about 1% of them even exceed 1.5ms, reaching a maximum of 2ms. When examining the head we notice that more than $\frac{1}{1000}th$ of the samples were less than 800μs. The meaning of this is that running a specific piece of code can vary in time by a factor of over 2.5! (0.78ms vs. 1.96ms).

The only interrupts that occured during the measurements were the timer and network interrupts. As

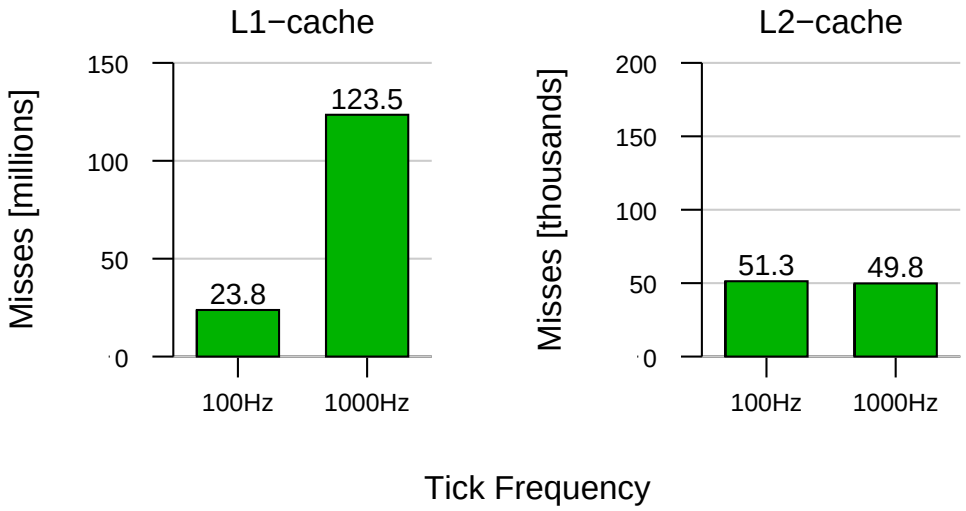

Figure 8: *Number of L1 and L2 cache misses, when running a 2.6.9 Linux kernel with 100Hz and 1000Hz timer frequency.*

Klogger and the application use the same cycle counter, we can identify repetitions that included specific interrupts and subtract them. However, Figure 7 also shows us that removing the direct overhead of these interrupts did not affect the measurement. Where did the cycles go, then?

The solution is apparent when measuring the cache misses caused by the interrupts. Figure 8 shows the number of cache misses caused by interrupts, when running the timer at both 100Hz and 1000Hz (100Hz is standard in the 2.4.x kernel) for perspective. It is clear that the number of cache misses caused by interrupts increases significantly with the increase in timer frequency, suggesting cache misses might cause the 1ms loop overhead. And indeed, when repeating the previous measurements with both the L1 and L2 caches disabled (Figure 9), subtracting the direct overhead leads to consistent measurements, indicating that the variability in the original measurements resulted from indirect overhead due to cache interference.

Identifying system noise is becoming a real problem for parallel and distributed computing [23]. This case study shows how Klogger's tight coupling with the underlying hardware can be used to pinpoint the computational noise generated by commong operating system interrupts.

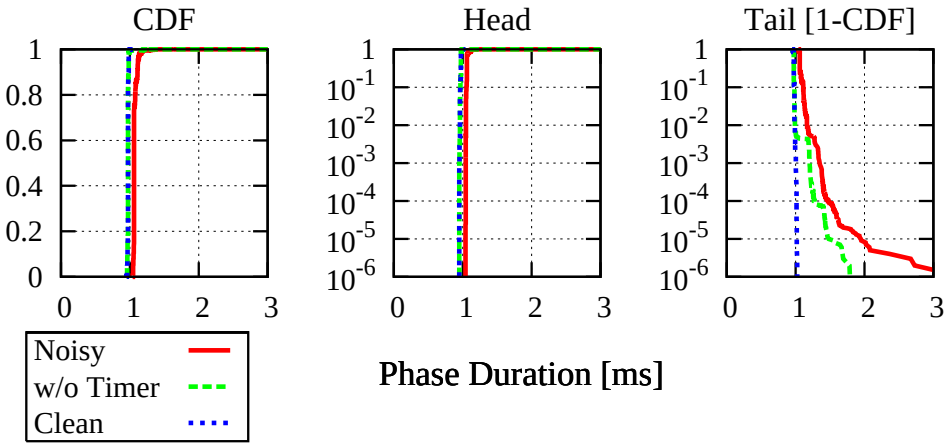

Figure 9: *The measurements of Figure 7 repeated with caches disabled. The graphs illustrate the raw intervals ("Noisy"), the intervals after removing the direct overhead of the timer ticks ("w/o Timer"), and the intervals with all interrupts' direct overhead removed ("Clean").*

Keeping Time in the Kernel

Operating systems keep track of time using the standard *8253 programmable interrupt timer* (PIT). PIT has been used in generations of processors for over 10 years.

In principle, whenever the kernel needs the wall clock time, it can simply access the 8253 through its I/O bus and read the data. This is done from the *do_gettimeofday* kernel function (of which the *gettimeofday* system call is just a wrapper). Reading the time from the 8253 PIT is a relatively expensive operation, so Linux is optimized (on machines which have a hardware cycle counter) to accesses the 8253 on every timer interrupt, and interpolate using the cycle counter in *do_gettimeofday*. Accessing the hardware's cycle counter is much faster than accessing the 8253 PIT, so this mode of operation limits the overhead incurred by the PIT to the number of timer interrupts per second. The two modes, common to both the 2.4.x and the 2.6.x kernel series, are called *PIT* mode and *TSC* mode.

Using Klogger's interrupt schema we have measured the overhead of the timer interrupt handler in both modes, on various generations of Intel processors. The results (originally presented in [11]) are shown in Figure 10. When running the kernel in PIT mode, the timer interrupt handler does not access the 8253 PIT. It consumes roughly the same number of cycles over all the hardware generations, so its μs overhead decreases as the hardware speed increases. When running the kernel in TSC mode, however, the 8253 is accessed from the timer interrupt handler. As access time to the PIT has not changed over the years, the time

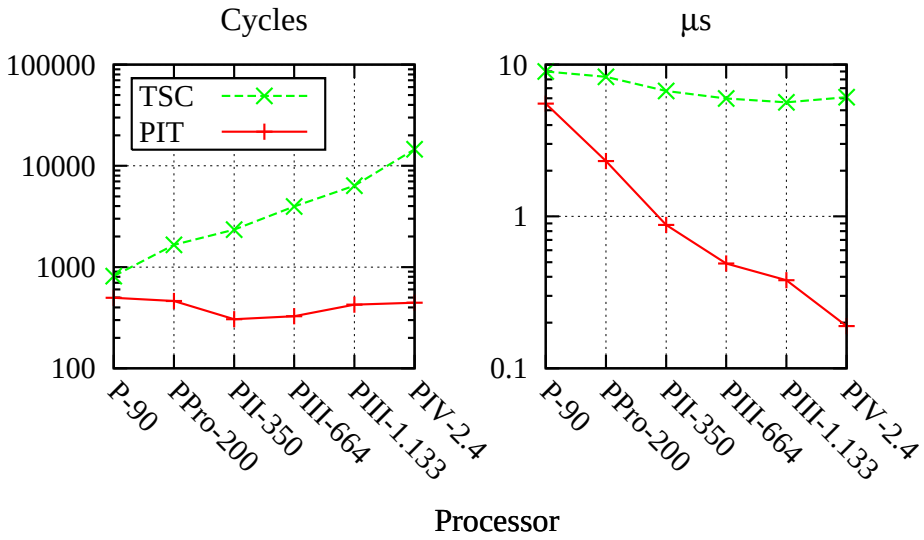

Figure 10: *Overhead of the timer interrupt when using the TSC and PIT modes. Note that the Y axis is log scaled.*

consumed by the handler remain rougly the same, and the number of cycles actually grows [21].

Given that TSC mode is the default, the timer interrupt handler is in fact becoming a liability — the more so as the timer interrupt rate increases (2.4 used 100Hz, whereas 2.6 uses 1000Hz). The TSC optimization, aimed at reducing the overhead of the *gettimeofday* system call, is actually workload dependant. It only helps for workloads that call *gettimeofday* at a higher frequency than the kernel's timer frequency. The solution should be accessing the 8253 PIT every *few* timer interrupts, preferably at a frequecy settable by the administrator.

These results demonstrate why measurements and tools are needed for kernel development. One kernel developer changed something in the kernel (the timer frequecy) but is unaware of its effect on another mechanism (the *gettimeofday* optimization). A simple performance measurement tool such as Klogger can help uncover such cases, allowing for more informed design decisions.

Conclusion and Future Work

We have presented *Klogger*, a low overhead, fine grained logging tool for the Linux kernel. Klogger's prime objective is to help analyze kernel behavior, and help researchers and developers understand what is really happening under the operating system's proverbial hood. Such support is required due to the increasing

number of developers working on the Linux kernel, and situations in which modules are inter-dependent in unexpected ways.

Making efficient use of the underlying hardware features, Klogger is able to achieve much finer granularity and lower overheads than any other kernel logging tool. Klogger's fine granularity and flexibility enables it to be used in the tightest corners of the kernel, and shed light on the operating system's nuts and bolts. Moreover, Klogger allows devlopers to create subsystem-specific logging schemata that can be used out-of-the-box by others. Another of Klogger uses is for kernel debugging. Although not discussed in the case studies, we have found it to be a very efficient debugging tool.

Using Klogger and its schemata in our research has helped us understand some interesting and sometime unexpected phenomena in the kernel. These case studies, which are the bulk of this paper, both demonstrate the tool's abilities, and more importantly suggest some major design problems in the Linux kernel. We have shown how locking issues can seriously limit the kernel's ability to handle SMP environments, and how both its scheduler and timing services' parameters are less than optimal for modern hardware.

Klogger however, does not exist in vacuum. Kernel developers and the research community should aspire to better understand the operating system kernels' intricacies. We hope a tool such as Klogger would be integrated into operating system development process by having developers write performance analyzing schemata for the subsystems they code. Klogger, its manual and the schemata described in this paper are available for download at *www.cs.huji.ac.il/labs/parallel/klogger*. It is our hope kernel researchers and developers will use this tool and create schemata for other subsystems — such as the filesystem, network, and others — through which we can all share our insights about the operating system's kernel operation.

Klogger is currently used by a few of our research colleagues, who provide us with feedback about it interface and capabilities. The reviews so far are very encouraging.

Although the current version of Klogger is Linux based, we hope to port it to other open source operating systems, such as FreeBSD. As Klogger only relies on a limited set of operating system atoms — kernel threads, *sysctl*, *test_and_set_bit* to name a few — porting it is a feasible task, enabling comparison between subsytems and mechanisms in different operating systems.

References

[1] J. M. Anderson, W. E. Weihl, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, and C. A. Waldspurger. Continuous profiling: where have all the cycles gone? In *ACM Symp. on Operating Systems Principles*, pages 1–14, 1997.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *ACM Symp. on Operating Systems Principles*, 2001.

[3] D. L. Black, A. Tevanian, D. B. Golub, and M. W. Young. Locking and reference counting in the Mach kernel. In *Intl. Conf. on Parallel Processing*, volume 2, pages 167–173, 1991.

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

[5] Z. Brown. What's new in kernel developement. *Linux Journal*, page 10, Mar 2005.

[6] R. Bryant, R. Forester, and J. Hawkes. Filesystem performance and scalability in Linux 2.4.17. In *Usenix Annual Technical Conf. (FreeNix Track)*, 2002.

[7] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the Linux kernel. In *4th Annual Linux Showcase & Conf.*, 2000.

[8] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Usenix Annual Technical Conf.*, June 2004.

[9] J. Casmira, D. Kaeli, and D. Hunter. Tracing and characterization of NT-based system workloads. *Digital Tech. J.*, 10(1):6–21, Dec 1998.

[10] Y. Endo and M. Seltzer. Improving interactive performance using TIPME. In *Intl. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 240–151, June 2000.

[11] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes. In *Intl. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 172–183, Jun 2003.

[12] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *IEEE Intl. Symp. on Signal Processing and Information Technology*, Rome, Italy, December 2004.

[13] Intel Corp. *IA-32 Intel Achitecture Software Developr's Manual. Vol. 3: System Programming Guide*.

[14] M. B. Jones and J. Regehr. The problems you're having may not be the problems you think you're having: Results from a latency study of windows NT. In *7th Workshop on Hot Topics in Operating Systems*, page 96, March 1999.

[15] M. Kravetz and H. Franke. Enhancing the Linux scheduler. In *Ottawa Linux Symp.*, 2001.

[16] J. Lions. *Lions' Commentary on UNIX 6th Edition*. Annabooks, 1996. (reprint).

[17] R. Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.

[18] R. Love, L. Trovalds, A. Cox, and various kernel developers. LKML Thread: Improving interactivity. http://kerneltrap.org/node/603, Mar 2003.

[19] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb 1991.

[20] S. W. Melvin and Y. N. Patt. The use of microcode instrumentation for development, debugging and tuning of operating system kernels. In *Intl. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 207–214, 1988.

[21] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Usenix Annual Technical Conf. (Summer)*, Jun 1990.

[22] P. S. Panchamukhi. Kernel debugging with Kprobes. http://www-106.ibm.com/developerworks/linux/library/l-kprobes.html, Aug 2004.

[23] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing*, Nov 2003.

[24] M. Rosenblum, E. Bugnion, S. Devine, , and S. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. on Modelling & Computer Simulation*, 1997.

[25] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, , and A. Gupta. The impact of architectural trends on operating system performance. In *ACM Symp. on Operating Systems Principles*, 1995.

[26] C. Schimmel. *UNIX Systems for Modern Architectures*. Addison Wesley, 1994.

[27] B. A. Schroeder. On line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78, June 1995.

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

[29] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *3rd Symp. on Operating Systems Design & Impl.*, Feb 1999.

[30] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Arch. Support for Programming Languages & Operating Systems*, pages 162–174, Oct. 1992.

[31] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *ACM Symp. on Operating Systems Principles*, Nov 1994.

[32] Various Kernel Deveoplers. LKML Thread: My thoughts on the ”new development model”. http://www.lkml.org, Oct 2004.

[33] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Usenix Annual Technical Conf.*, June 2000.