Feather-Trace: A Light-Weight Event Tracing Toolkit*

Björn B. Brandenburg and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

We present a light-weight event tracing toolkit for real-time operating systems on the Intel x86 platform. Our approach is wait-free, multiprocessor-safe, and introduces very low overhead. Only a single unconditional jump instruction is required to distinguish between enabled and disabled events. As a case study, we traced the locking behavior of the Linux kernel and several soft real-time multimedia applications. Our results provide strong support for the wide-spread assumption that short non-nested critical sections are the common case in practice.

1 Introduction

When developing operating systems and embedded systems, *event tracing facilities* are an essential tool. Such facilities allow developers to trace the behavior of the system being developed by collecting performance and state data while the system in question executes for later offline analysis. The ability to better understand observed behaviors and to obtain high-resolution timing information greatly helps to both debug failures and improve performance. Thus, it is not surprising that there has been considerable recent interest in tracing frameworks [5, 7, 11, 19, 20].

Prior work. For general-purpose operating systems, powerful and flexible solutions have been developed and integrated into commercially-available products. For example, the DTrace facility of the Solaris 10 operating system, developed by Sun Microsystems [5], offers flexible dynamic instrumentation support. By embedding a virtual machine inside the kernel, it allows event data to be safely gathered and processed at arbitrary locations inside the kernel by compiled trace scripts. Such flexibility comes at a price, however. The DTrace implementation is complex and requires many operating-system services such as run-time symbol information, which may not be present in (space-constrained) embedded systems. Further, interrupts are disabled while executing trace scripts, which makes it unfit for use in real-time systems. Other dynamic instrumentation approaches based on binary re-writing such as kerninst [12] also require substantial in-kernel infrastructure. The K42 kernel [8] provides a lock-free, unified performance monitoring facility. While it provides a high-performance event tracing facility, its implementation is closely tied to the memory-management implementation of the K42 operating system, and thus cannot be easily ported to other operating systems. Disabled events incur an overhead of four instructions [19], some of which access main memory and affect branch prediction. Also, the use of potentially unbounded lock-free retry loops in K42's buffer implementation may restrict its applicability in hard real-time environments. The Ferret framework has been designed specifically for the Dresden Real-Time Operating System Project (DROPS) [10] and is based on a rather heavy-weight architecture. It is designed to allow tracing of real-time and best-effort tasks, system services, and the microkernel. However, the reliance on an event-structure description language and a custom tool chain restricts the portability of the framework. While tools that capture instruction-level execution traces such as Nirvana [1] provide a wealth of information for offline analysis, their use for obtaining real-world timing information is limited due to high overheads.

Motivation and contributions. In this paper, we present a light-weight, multiprocessor-safe tracing toolkit called *Feather-Trace*. Our motivations in producing this toolkit were two-fold. First, our research group has been engaged in an ongoing development effort involving a system called LITMUS$^{\mathrm{RT}}$ [3, 4, 17], which extends the base Linux kernel so that different scheduling and synchronization methods can be loaded as plug-in components. The primary focus of our LITMUS$^{\mathrm{RT}}$-related research has been scheduling and synchronization support for multiprocessor real-time systems. Our

*Work supported by a grant from Intel Corp., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425. The first author was also supported by a Fulbright fellowship.

current development platform for LITMUS$^{\mathrm{RT}}$ is a four-processor machine. In order to debug scheduling and synchronization code in LITMUS$^{\mathrm{RT}}$, we needed a tracing mechanism that could be used on a multiprocessor with very low overhead, and that could be invoked anywhere in the kernel. We found that existing tracing mechanisms were ill-suited for our purposes. Second, in devising and evaluating synchronization mechanisms implemented in LITMUS$^{\mathrm{RT}}$ [2, 3], we desired to have a better understanding of locking patterns that are typical of "real-world" systems, so that we could optimize these mechanisms for common-case scenarios. Linux itself is certainly a real-world system, so we desired to trace its behavior to assess the frequency, duration, and degree of nesting in lock accesses. To validate our trace data, we also instrumented several soft real-time multimedia applications. Again, we found existing tracing facilities to be unsuitable for our purposes. Rather than providing a complete tracing framework, we found that our needs were best met by a highly-portable toolkit that can be easily integrated into existing operating systems with some "glue code." In Feather-Trace, trace events are checked via a single unconditional jump instruction, and trace data is collected in wait-free buffers that can be efficiently accessed on different processors. Although we were motivated by the specific concerns just noted in producing Feather-Trace, because it is very light-weight, can be used anywhere in the kernel, and it is portable, it should be of use to others engaged in kernel-related research. To the best of our knowledge, Feather-Trace is the first static tracing toolkit that achieves a single-instruction overhead in the case of both enabled and disabled tracing events.

The rest of this paper is organized as follows. In Section 2, we present Feather-Trace. In Section 3, as a case study, we present some measurements of the locking behavior of the Linux kernel and several soft real-time applications. Finally, in Section 4, we conclude.

2 Feather-Trace

To trace the execution of an operating system, a toolkit must provide methods to embed "triggers" in the program text and to collect data for offline analysis. The purpose of a *trigger* is to redirect the flow of execution to a user-provided *callback function* that can take appropriate actions such as collecting performance data or checking invariants for debugging purposes.

To be of practical use, several requirements must be met. First, it should be possible to selectively *enable* and *disable* triggers, since it is likely that only a specific aspect of an operating system is being inspected at any time. Second, no assumption concerning the execution context and preemptivity should be made so that triggers can be placed anywhere in the kernel, including interrupt handlers. Third, the framework should be multiprocessor-safe and it should not introduce additional mutual-exclusion requirements—by "multiprocessor-safe," we mean that tracing actions on one processor should not adversely affect other processors. Further, to increase portability and suitability for embedded platforms, only very little support should be required from the operating system. Of course, any overheads introduced by the tracing framework must be kept at a minimum. This implies that the trigger code should be short and affect neither cache performance nor the processor's branch prediction accuracy negatively. Ideally, a disabled trigger should incur no costs.

2.1 Event Trigger

In Feather-Trace, event triggers are realized as C preprocessor macros (`ft_eventX()`, where `X` is the number of arguments) that insert trigger code realized with inline assembly instructions. Thus, as is always the case with static instrumentation, events can only be added at compile time. While this may be an unacceptable limitation in the case of general-purpose operating systems such as Solaris, dynamic instrumentation has the disadvantage that enabled events incur the (considerable) costs of a CPU exception [10]. Also, if a custom exception handler were to be required, then adding tracing to an existing operating system would require intrusive modifications of its exception-handling code, thereby drastically increasing development effort.

The trigger code must accomplish three tasks. First, it must determine whether the event is enabled. If it is enabled, it must collect the necessary context information and invoke the callback function associated with the event. Finally, it must restore the processor context so that the original code surrounding the trigger can proceed correctly.

To achieve the goal of negligible overhead, the decision whether to invoke the callback function must be made as quickly as possible. Therefore, we chose the following approach (illustrated in Fig. 1): the `ft_eventX()` macro precedes the invocation of the callback function with an unconditional jump instruction (`jmp`) that skips over the rest of the trigger code. Thus, events are initially disabled. To enable an event, the offset parameter of the jump instruction is set to zero, which effectively disables the jump. As a result, the required context information is pushed on the stack and control is transferred to the callback function.

Since the trigger code is less than 128 bytes long, in

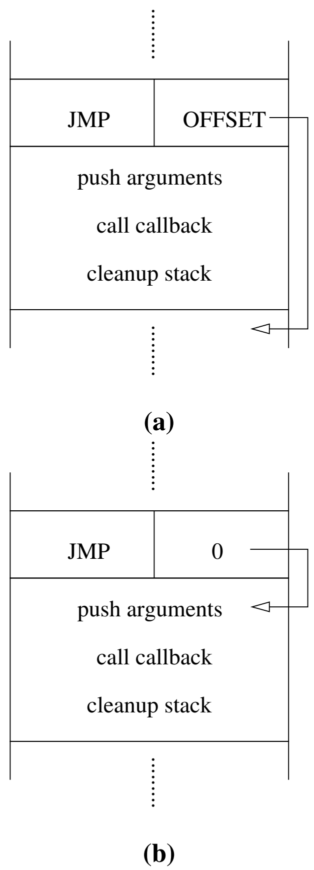


Figure 1: An illustration of the trigger assembly code. **(a)** In the disabled state, the jump instruction will skip the invocation of the callback. **(b)** In the enabled state, the jump instruction's offset is zero.

the Intel x86 instruction set, the unconditional jump including the offset can be encoded in two bytes. The jump instruction code in the first byte (`0xeb`) is followed by a signed eight-bit integer, which is the offset of the desired destination. To enable or disable an event, only the offset must be changed. Since eight-bit write operations to arbitrary byte-aligned addresses are guaranteed to be atomic on the Intel x86 platform, enabling and disabling events is multiprocessor-safe.

To summarize: events can be safely enabled and disabled on multiprocessors. No operating-system support is necessary and no locking/mutual-exclusion support is required. If an event is disabled, then only one additional instruction is executed compared to the case if there were no trigger code present. On the other hand, if an event is enabled, then only one additional instruction is executed compared to a normal function call. Determining whether a given event is enabled with only a single instruction that does not access memory (and which also has no effects on either branch prediction or pipelining)

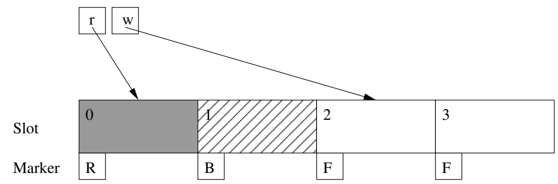


Figure 2: An illustration of a wait-free bufer for $n = 4$ and $f = 2$. Slot 0 is ready, Slot 1 is busy and being written, Slots 2 and 3 are free. The current read index r points to the next ready slot (Slot 0), and the current write index w points to the next free slot (Slot 2).

in both the enabled and the disabled case is arguably optimal. An additional small overhead can be incurred because the compiler may be forced to (re-)load some registers before and after the trigger code. This effect can be reduced by placing triggers mainly at the start and end of functions.

2.2 Data Collection

A tracing framework is of no utility if it does not offer a method to collect data. To keep the overhead of enabled events low, any trace data should be temporarily accumulated in an in-memory buffer and be transferred to stable storage after a certain number of samples have been obtained. To support multiprocessors, such a buffer must allow for multiple concurrent writers and, for similar reasons as is the case for triggers, should not rely on mutual exclusion to achieve correctness. While read operations should be possible in parallel with write operations, there is usually no great need for multiple readers, since typically a single reader is tasked with flushing the buffer to stable storage.

To attain the stated goals, Feather-Trace provides a *wait-free* FIFO-buffer implementation to store event data. The buffer is said to be wait-free since no locks are required and each read and write operation completes in a bounded number of steps (such is not the case when lock-free retry loops are used). Our implementation supports arbitrarily many concurrent writers. To simplify the data structure and to improve performance, we allow only one concurrent reader.

As illustrated in Fig. 2, a buffer consists of n *slots*. Slots may be of arbitrary but uniform size s. Each slot is associated with a *slot marker* that indicates the current state of the slot. A slot may be either *free*, *busy*, or *ready*. For each buffer, the number of free slots f (a signed 32-bit integer), the current write index w, and the current read index r (both unsigned 32-bit integers) are maintained. We require that n divides the maximum value that an unsigned 32-bit integer can store, *i.e.*,

```
unsigned int r = 0, w = 0, e = 0;
int f = n;

start_write(void **ptr) {
  unsigned int idx;
  if (fetch_and_dec(f) <= 0) {
    /* buffer full */
    atomic_inc(f);
    atomic_inc(e);
    *ptr = NULL;
    return 0;
  } else {
    /* slot reserved */
    idx   = fetch_and_inc(w) % n;
    marker[idx] = SLOT_BUSY;
    *ptr = &slot[idx];
    return 1;
}

finish_write(void *ptr) {
  unsigned int idx;
  idx = (ptr - &slot[0]) / s;
  marker[idx]   = SLOT_READY;
}

read(void* dest) {
  unsigned int idx;
  if (f == n)
    /* nothing available */
    return 0;
  idx = r % n;
  if (marker[idx] == SLOT_READY) {
    memcpy(dest, &slot[idx], s);
    marker[idx] = SLOT_FREE;
    r++;
    atomic_inc(f);
    return 1;
  } else
    return 0;
}
```

Figure 3: Pseudo-code for the methods used to access the wait-free buffers provided by Feather-Trace.

$2^{32} \bmod n = 0$. This allows us to ignore integer overflows, which is a minor performance improvement.

To detect missed samples, the number of failed writes is stored in the error count e. (A write fails if the buffer is full.) Pseudo-code for the buffer access methods is given in Fig. 3. The implementation relies on the atomic `XADD` ("exchange and add") instruction, which is used to realize `fetch_and_dec/inc()` and `atomic_inc()`. Writers access the buffer by first invoking `start_write()` to obtain a pointer to a free slot. A slot is reserved in two steps. First, the number of free slots f is read and decremented atomically to reserve a slot. If a reservation can be made ($f > 0$ holds), then the next free slot is obtained by atomically reading and incrementing w. Since n divides 2^{32}, a potential overflow of w does not need to be handled. The slot is marked as busy to prevent a concurrent reader from observing incomplete data. If no slot is available, then the reservation is canceled by atomically incrementing f and the error count e. The single reader accesses the buffer by first checking whether there exist non-free slots by comparing f and n. If there exists such a slot, then the reader checks the slot's state, and if the slot is ready, copies the slot's contents to a reader-provided location such as I/O buffers.

The multi-writer, single-reader, wait-free FIFO buffer provided by Feather-Trace offers a low-overhead method to store uniformly-sized data items. The limitation of uniformly-sized items can be easily dealt with by providing several buffers of different sizes. As both the provided event triggers and FIFO buffers are designed to minimize overheads, Feather-Trace can be used to trace highly performance-critical code sections. For example, as explained in more detail in the next section, we have used the toolkit to measure critical section lengths in the Linux kernel. This was made possible in part because the code used to obtain and store time stamps, including the event trigger, consists of only 61 instructions, which is a negligible overhead in most cases.

Since the toolkit is minimally intrusive and makes no assumptions on the availability of operating-system services, it can be easily integrated into existing code bases. For example, we have used Feather-Trace to obtain event traces in both the Linux kernel and the FreeBSD kernel by implementing a custom device driver that exports the accumulated event data to user space. Further, by pre-loading Feather-Trace (packaged in a shared library) into dynamically linked user space applications, we were able to record the locking behavior of various soft real-time multimedia applications.

3 Case Study: Locking in Linux

One motivation for the development of Feather-Trace was to allow us to obtain empirical results on the frequency, degree of nesting, and duration of critical sections in "real-world" systems. In prior work, Devi *et al.* [6] measured the length of critical sections accessing common data structures in order to generate task sets for schedulability-analysis purposes [6]. The method employed by them, however, cannot give insight into the nesting depth and the distribution of lock requests, as it relies on measuring synthetic tasks. Other studies have assessed the impact of lock-free synchronization on large scientific applications [15, 16]. Unfortunately, these benchmarks are mostly concerned with overall performance and do not reveal the nature of individual critical sections. In this paper, we seek to provide additional data points on "real-world" locking behavior by measuring critical sections in both the Linux kernel under various workloads and several soft real-time applications.

In the following subsections, we say that a lock request has a *nesting depth* of n if the processor already was holding n locks at the time of the request. Further, we define the *critical section length* of a lock to be the length of the time interval that starts when the lock is successfully acquired and ends just before it is released again, *i.e.*, the cost of acquiring the lock itself is not included.

3.1 Kernel Modifications

We modified the Linux kernel, version 2.6.20, to capture timing information on critical section lengths. The Linux kernel employs two different kinds of locks, *spinlocks* (contention is handled by busy-waiting) and *semaphores* (processes are suspended in case of contention). To trace spinlocks, we changed the locking primitives `spin_lock()`, `read_lock()`, `write_lock()`, and the corresponding unlock primitives (as well as special cases such as `spin_lock_irqsave()`) to include event triggers after a lock has been acquired and before a lock is released. To trace semaphores, we modified `mutex_lock()`, `down()`, and related primitives such as `down_read()` in a similar fashion. At each event, a time stamp was obtained by reading the `TSC` register (which can be read from both user and kernel space) and the sample consisting of the time stamp, the CPU on which the event occurred, the address of the lock involved, and the type of the event (enter critical section, or exit critical section) was stored in a Feather-Trace buffer. That buffer was made available to user space by means of a custom character device driver.

3.2 Setup of the Experiments

To obtain insight into the kernel's locking behavior, we executed various test workloads and captured locking events during several intervals of 60 seconds each. Our particular test platform is an SMP consisting of two 32-bit Intel(R) Xeon(TM) processors running at 2.70 GHz, with 8K instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory.

The results of three workloads are presented in this paper. First, we obtained a trace from an otherwise idle system. Second, we traced the locking behavior of the Linux kernel while compiling a copy of the kernel itself. Last, we used the `stress` utility [18] to generate a test load of three processes that stressed the memory management and I/O subsystems of the kernel.

The captured event traces were analyzed offline as follows. After filtering incomplete event sequences (*i.e.*, lock accesses that were missing one of the two expected timestamps), the remaining lock requests were annotated with their respective nesting depth. Incomplete sequences may occur at both the beginning and the end of the trace interval and when there is insufficient buffer space available. The clock speed of the processors (2.70 GHz) was used to convert raw cycle-count timestamps to microseconds. Finally, histograms of the nesting level and the critical section length (with a bin size of 0.1μs), the cumulative distribution, and the average critical section length were computed for both spinlocks and semaphores.

3.3 Results

After filtering, the traces contained valid events for a total of 2,366,122 (idle system), 25,002,249 (compile test), and 70,807,495 (stress test) spinlock and 51,360 (idle system), 4,880,570 (compile test), and 18,998,386 (stress test) semaphore acquisitions.

The distribution of the spinlock nesting depth is shown in Fig. 4. The maximum nesting depth in the presented data is four under load and three on an idle system. One can clearly see that the vast majority of lock acquisitions are non-nested, *e.g.*, under load, more than 85 percent of all lock requests have a nesting depth of zero. When comparing the nesting depth distribution of an idle system to the distribution observed during the stress test, one can see two trends. First, maximum nesting depth increases from three to four (nesting depths as deep as six have been observed in traces not presented here, but occur so rarely that they are hard to reproduce), which reveals that there exist deeply nested lock requests that are only required very seldomly. Second, the percentage of non-nested lock requests increases under load, which can be attributed to the fact that the number of shared objects that are mostly accessed in a non-nested fashion increases under load. As can be seen in Fig. 6, critical sections protected by semaphores are less frequently subject to nesting as those protected by spinlocks. Semaphore requests exceeding two levels of nesting were not observed in any of the traces.

In Fig. 5, the distribution of spinlock critical section lengths is depicted. While the distributions do have a long tail, more than 96 percent of all observed critical sections are shorter than 5μs. Inset (a) depicts the distribution of an idle system. Since system-call activity is low, most lock requests are issued by interrupt handlers. One can clearly see two distinctive spikes as a result, because the critical sections encountered in periodic activities such as the timer-interrupt service routine contribute the majority of observed critical sections. The average critical section length observed in an idle

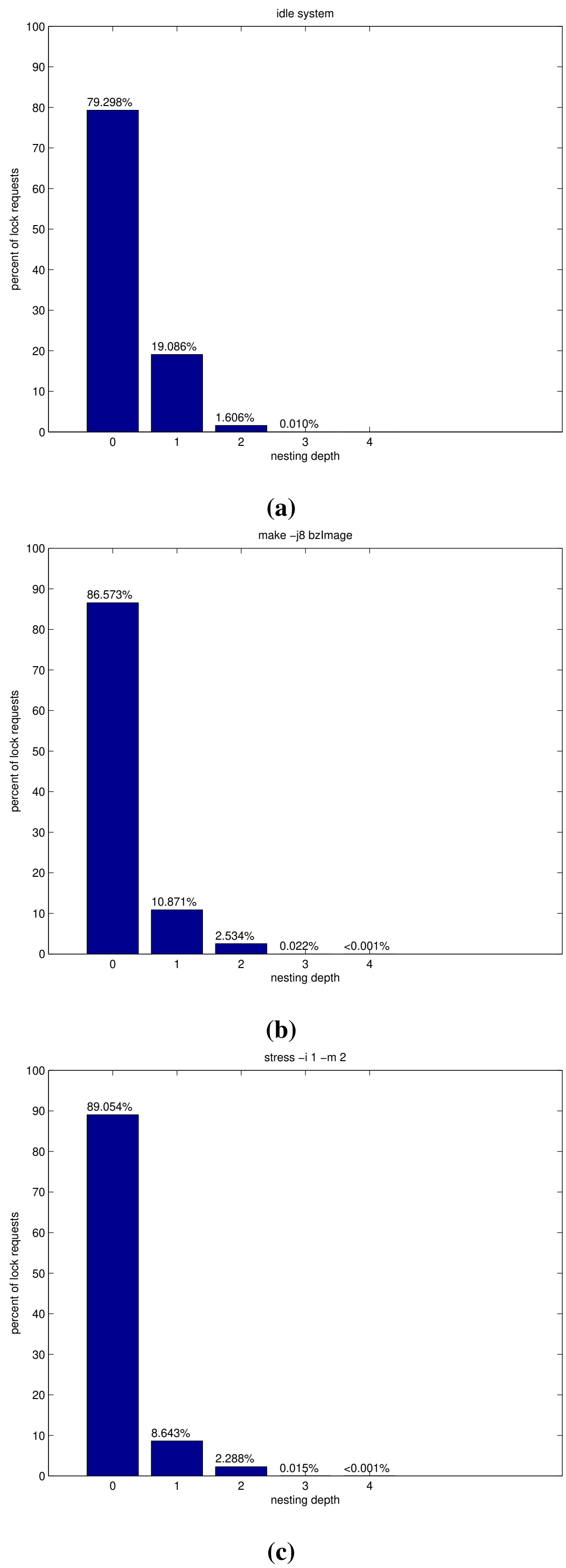


Figure 4: Distribution of nested spinlock accesses in the Linux kernel under various work loads.

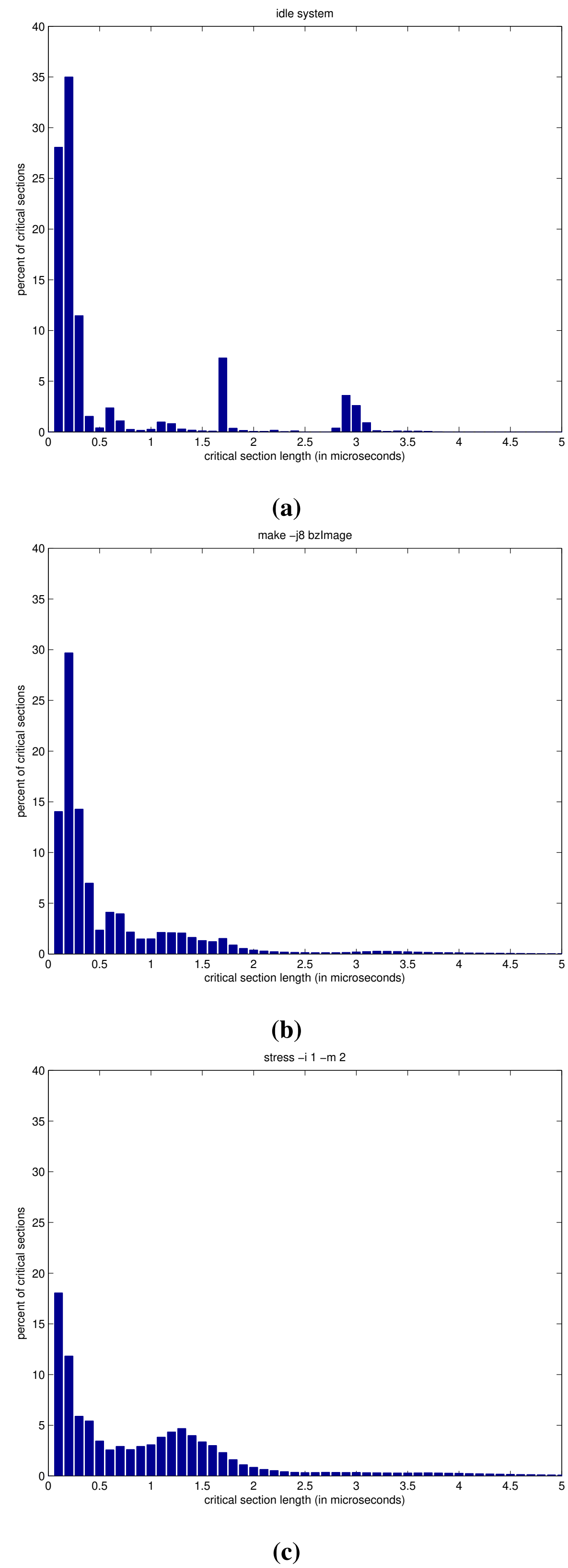


Figure 5: Distribution of spinlock critical section length in the Linux kernel. More than 96 percent of all observed critical sections were shorter than $5\mu s$.

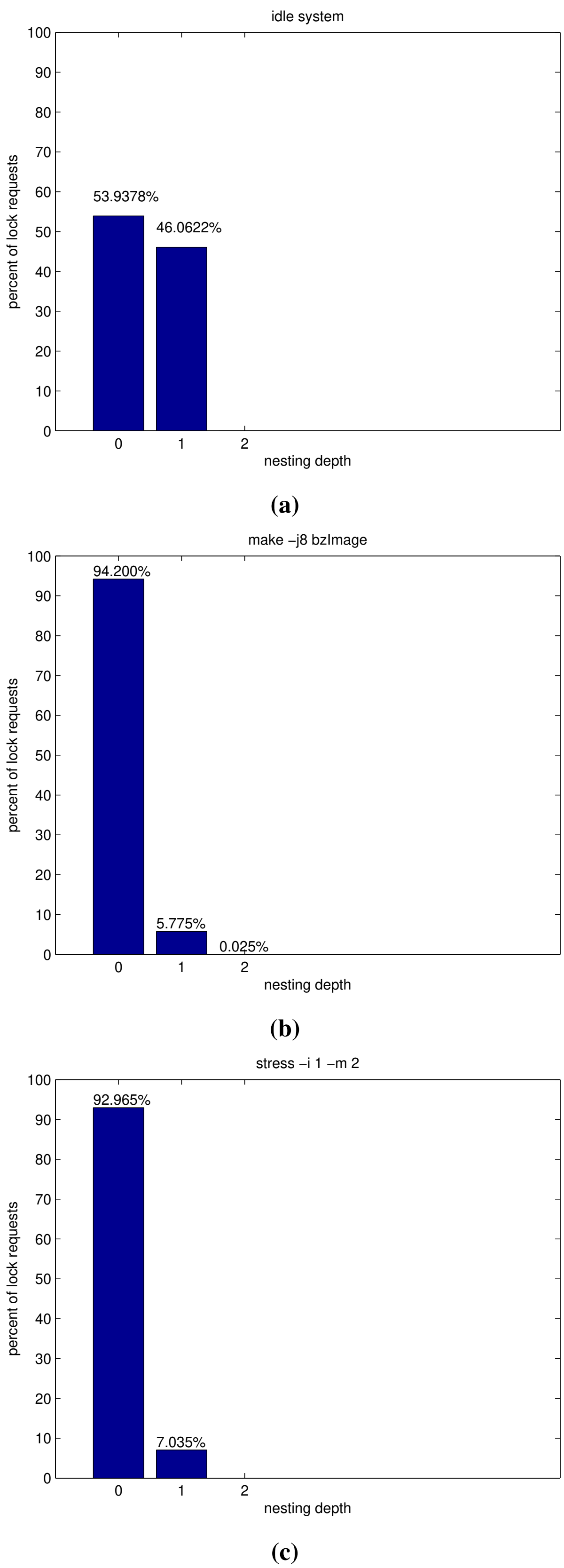


Figure 6: Distribution of nested semaphore accesses in the Linux kernel under various work loads.

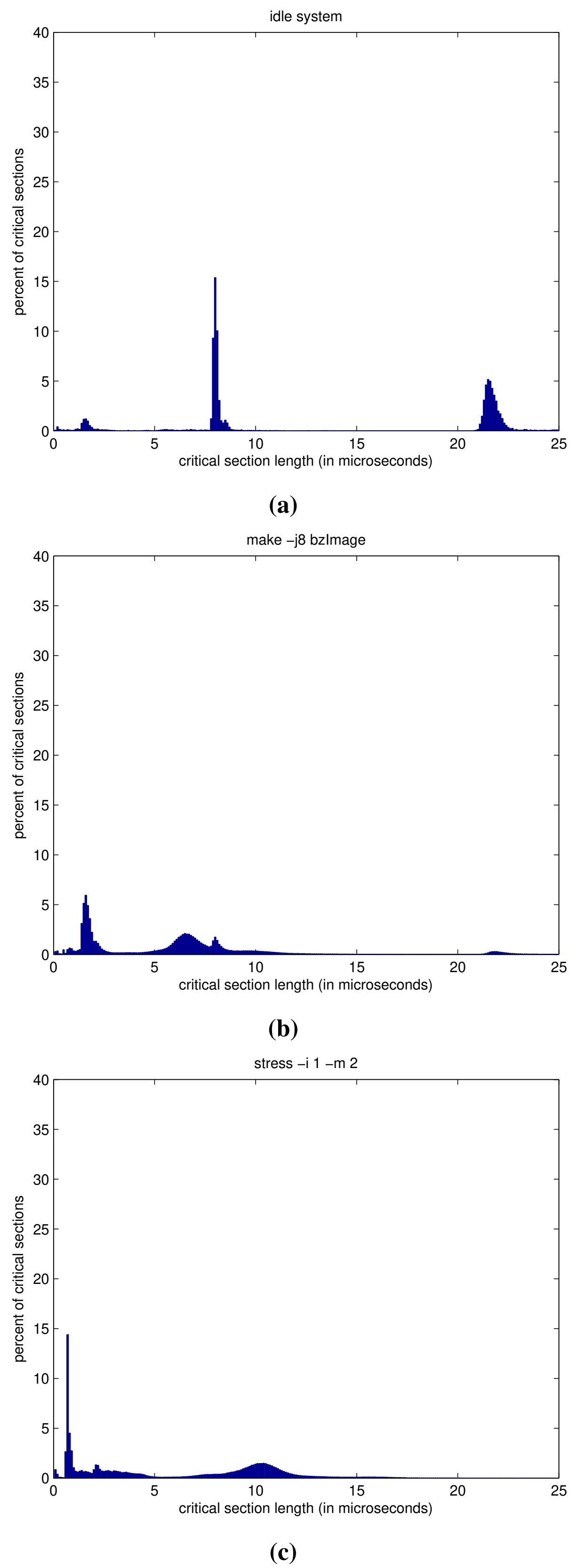


Figure 7: Distribution of semaphore critical section length. Under load, more than 93 percent of all observed critical sections were shorter than $13\mu s$.

system was 0.67μs. The distribution of critical section lengths observed in the compile benchmark is shown in inset (b). The impact of the periodic activities decreases noticeably compared to an idle system. As a result of the kernel actually doing "real" work on behalf of user-space processes, the critical section lengths are spread out over a wider range. The average critical section length observed in this benchmark increased to 0.81μs. The trend continues in inset (c), which depicts the distribution observed under the stress test. In this case, the kernel has to service many "expensive" system calls, so that the center of the distribution is shifted noticeably to the right. The average observed critical section length was 1.24μs. In Fig. 7, the distribution of semaphore critical section lengths is shown. Critical sections protected by semaphores are typically significantly longer than those protected by spinlocks. The average observed critical section lengths were 6.3μs (stress test), 6.4μs (compile test), and 14.9 μs (idle system).

3.4 Soft Real-Time Applications

Since the Linux kernel may not be representative of real-time applications, we conducted similar experiments with several multimedia applications running on top of Linux 2.6.23-rc3 to ensure the validity of our conclusions. The results for three of the benchmarks are shown in Figs. 8 and 9. (Because the distributions did not contain characteristic spikes, we chose to present them as cumulative distributions instead.) With each tested application, we used Feather-Trace to instrument the acquisition and release of user space binary semaphores as provided by the POSIX thread (`pthread`) library. As was the case with the kernel, cycle-count timestamps were used to determine the beginning and end of a critical section. The data depicted in Insets (a) and (b) was obtained by instrumenting two popular open source video players (Video Lan Client (VLC) [13] and Xine [14]) over a period of one hour. Inset (c) shows the behavior of Tux Racer [9], an interactive 3D video game, over a period of about one minute. Since these applications need to ensure that both visual and audio content is presented to the user in a timely manner they can be considered to be soft real-time applications.

Fig. 8 clearly shows the nesting characteristics of the three applications. While nesting almost never occurs in Tux Racer, and only very rarely in Xine, it is used more commonly in the VLC video player. However, non-nested accesses, which make up more than 70 percent of the critical sections, are still the common case. A nesting level greater than three was never observed in the tested multimedia applications.

Distributions of critical section lengths are depicted in Fig. 9. As opposed to the nesting levels, the cumulative critical section length distributions of the instrumented applications are somewhat similar. In all cases, more than 95 (99) percent of the critical sections are shorter than 5μs (10μs). This indicates that critical sections in multimedia applications are typically even shorter than those observed in the kernel. This observations is also supported by a significantly lower average critical section length (compared to the average length of in-kernel semaphore-protected critical sections).

Our results strongly support the wide-spread assumption that the vast majority of critical sections in many settings are short and non-nested. While deep nesting does occur in practice, nesting depths of three or more occur only rarely. Critical sections longer than 5μs (13μs) are rare in the case of spinlocks (semaphores, under load) in the kernel. In multimedia applications, they tend to be even shorter. Thus, our data supports the claim that the common case in practice is short, non-nested lock requests.

4 Conclusion

This paper presented Feather-Trace, a new light-weight static tracing toolkit that is both highly portable and can be used for performance data collection as well as debugging purposes. Because Feather-Trace uses neither locks nor retry loops, it is suitable for hard real-time environments. Further, since disabled events incur only the negligible overhead of one additional instruction per event, there is no need to remove Feather-Trace in production releases.

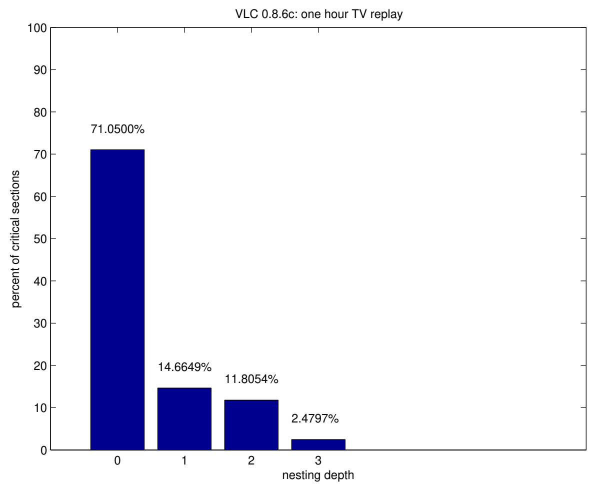
As a case study and to support our ongoing work on multiprocessor real-time synchronization, we used Feather-Trace to obtain the frequency, duration, and degree of nesting in lock accesses in both the Linux kernel under various workloads and soft real-time applications. Our measurements strongly support the wide-spread assumption that short, non-nested critical sections are by far the common case in practice.

As mentioned in the introduction, we believe that Feather-Trace may be of interest to a wider audience of embedded and real-time systems developers. Our implementation is available under a permissive open source license at the first author's home page[1].

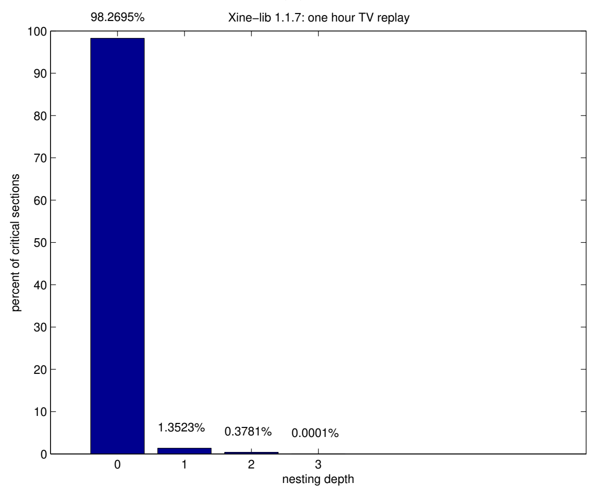
References

[1] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihoĉka, and J. Chau. Framework for instruction-level tracing and analysis of pro-

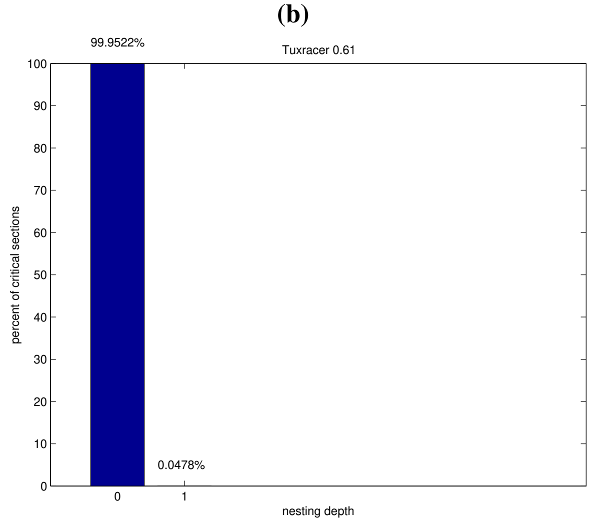
[1] http://www.cs.unc.edu/~bbb/feathertrace/ .



(a)

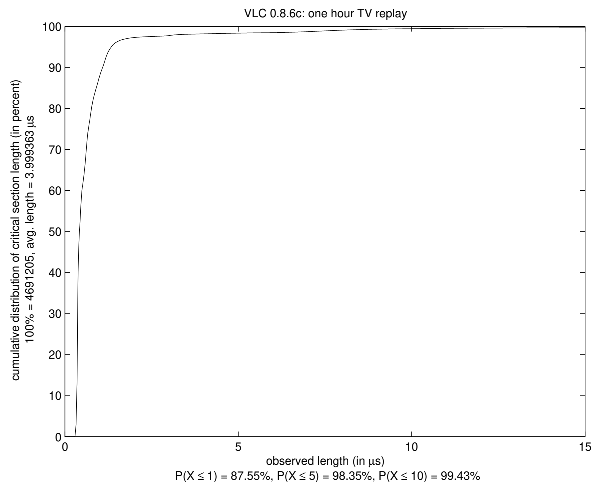


(b)

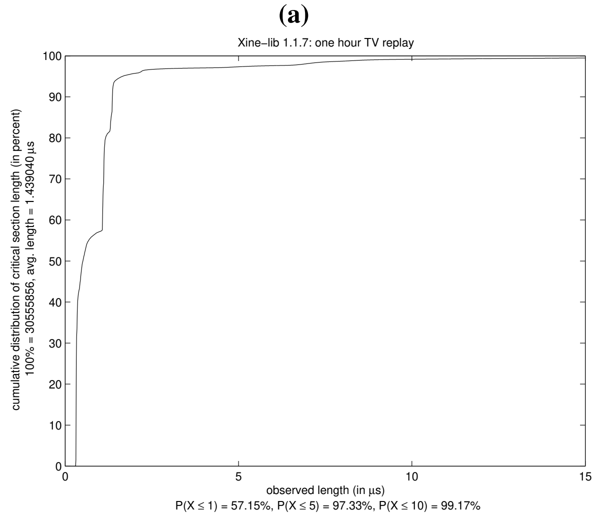


(c)

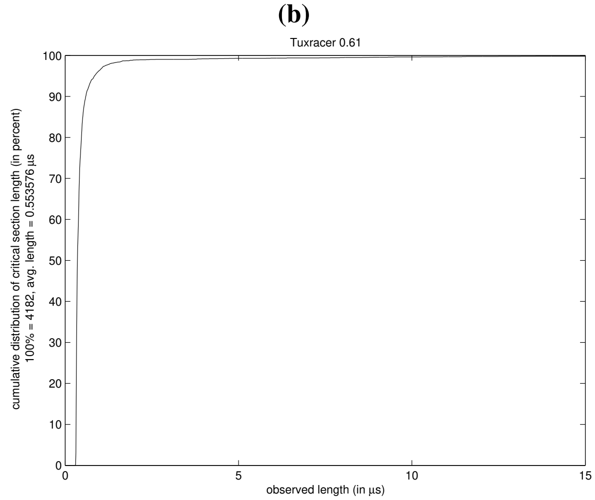
Figure 8: Distribution of nested mutex accesses in multimedia applications.



(a)



(b)



(c)

Figure 9: Distribution of mutex critical section length in multimedia applications. Note, that 99% of the critical sections were shorter than $10\mu s$.

gram executions. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, 2006.

[2] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2007. To appear.

[3] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In submission, 2007.

[4] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[5] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX '04*, 2004.

[6] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, 2006.

[7] IBM Linux Technology Center. Dynamic probes. Homepage. http://dprobes.sourceforge.net/.

[8] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of EuroSys 2006*, 2006.

[9] J. Patry. Tux Racer. Homepage. http://tuxracer.sourceforge.net/.

[10] M. Pohlack, B. Döbel, and A. Lackorzynski. Towards runtime monitoring in real-time systems. In *Proceedings of the Eighth Real-Time Linux Workshop*, 2006.

[11] Red Hat, IBM, Intel, and Hitachi. System tap. Homepage. http://sourceware.org/systemtap/.

[12] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[13] The VideoLan Project. VideoLan Client. Homepage. http://www.videolan.org/.

[14] The Xine Project. Xine Libraries and UI. Homepage. http://xinehq.de/.

[15] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, 2001.

[16] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proceedings of the the Third Int'l Workshop on Software and Performance*, 2002.

[17] UNC Real-Time Group. LITMUS$^{\mathrm{RT}}$project. Homepage. http://www.cs.unc.edu/˜anderson/litmus-rt/.

[18] A. Waterland. stress. Homepage. http:// weather.ou.edu/˜apw/projects/stress/.

[19] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of SC 2003*, 2003.

[20] K. Yaghmour. Linux trace toolkit. Homepage. http://www.opersys.com/LTT/.