Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems

Robert W. Wisniewski [‡] Bryan Rosenburg [‡]

Abstract

Programming, understanding, and tuning the performance of large multiprocessor systems is challenging. Experts have difficulty achieving good utilization for applications on large machines. The task of implementing a scalable system such as an operating system or database on large machines is even more challenging. And the importance of achieving good performance on multiprocessor machines is increasing as the number of cores per chip increases and as the size of multiprocessors increases. Crucial to achieving good performance is being able to understand the behavior of the system.

We have developed an efficient, unified, and scalable tracing infrastructure that allows for correctness debugging, performance debugging, and performance monitoring of an operating system. The infrastructure allows variable-length events to be logged without locking and provides random access to the event stream. The infrastructure allows cheap and parallel logging of events by applications, libraries, servers, and the kernel. The infrastructure was designed for K42, a new open-source research kernel designed to scale near perfectly on large cache-coherent 64-bit multiprocessor systems. The techniques are generally applicable, and many of them have been integrated into the Linux Trace Toolkit. In this paper, we describe the implementation of the infrastructure, how we used the facility, e.g., analyzing lock contention, to understand and achieve K42's scalable performance, and the lessons we learned. The infrastructure has been invaluable to achieving great scalability.

[‡]IBM T. J. Watson Research Center
This work was supported in part by a DARPA PERCS grant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC'03, November 15-21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011...$5.00

1 Introduction

The importance of being able to understand the behavior of a system in order to achieve good multiprocessor performance has long been understood. As multiprocessors have grown in complexity, understanding system behavior has become more important. Many operating systems did not contain in their original design a mechanism to understand performance. Many times, as those systems evolved, different tailored mechanisms were implemented to examine the portion of the system that needed evaluation. For example, in Linux, there's a device-driver tracing infrastructure, one specific to the file system, the NPTL trace facility, one-off solutions by kernel developers for their own code, plus more-general packages including oprofile, LKST, and LTT. Even commercial operating systems often had several different mechanisms for obtaining tracing information, for example, IRIX had three separate mechanisms. Some of these mechanisms were efficient, but often they were one-off solutions suited to a particular subsystem and were not integrated across all subsystems.

Part of the difficulty in achieving a common and efficient infrastructure is that there are competing demands placed on the tracing facility. In addition to integrating the tracing infrastructure with the initial system design, we have developed a novel set of mechanisms and infrastructure that allows us to use a single facility for correctness debugging, performance debugging, and performance monitoring of the system. The key aspects of this infrastructure are the ability to log variable-length events in per-processor buffers without locks using atomic operations, and given those variable-length events, the ability to allow random access to the data stream. The infrastructure when enabled, is low impact enough to be used without significant perturbation, and when disabled has almost no impact on the system, allowing it to remain always ready to be enabled dynamically.

The K42 project[2] is developing a new open-source operating system kernel incorporating innovative mechanisms and policies and modern programming technologies. K42 is designed to scale up to 64-bit machines with thousands of processors and down to ubiquitous 2- to 8-way multiprocessors. Our goal is to start with a "clean

slate" and examine the system structure needed to achieve excellent performance in a scalable, maintainable, and extensible system. Although we wanted to design from scratch, we could not, and did not want to, implement all aspects of an operating system from scratch. Further, requiring applications to use a new API would make experimenting with the system unpalatable to potential users. We therefore did not introduce a new personality, but instead made K42 fully Linux API- and ABI-compatible[1].

Because we designed the system from scratch, we were able to integrate tracing and performance monitoring from the earliest stages, allowing an efficient, unified, and scalable facility. Providing this integration in a mature operating system is possible, but tends to be complicated by individual developers having grown accustomed to the tracing/analysis mechanisms of their subsystems. K42's infrastructure allows cheap and parallel logging of events by applications, libraries, servers, and the kernel. This event log may be examined while the system is running, written out to disk, or streamed over the network. Post-processing tools allow the event log to be converted to a human readable form or to be displayed graphically. Lockless logging is a key feature yielding very efficient tracing, and the unified infrastructure allows the facility to remain compiled into the system.

We have written a set of trace analysis tools, including one designed to allow graphical display of the data. While the tools are not the focus of this paper they demonstrate the power the data can provide. The description of how we use them also provides insight into how we tuned K42 to achieve good scalable performance. One of the tools allowed us, for example, to evaluate lock contention on an instance by instance basis. It is hard to understate the importance of being able to graphically view the data. This capability has allowed us to observe performance difficulties that would otherwise have gone undetected. In addition to performance tuning, as we have mentioned above, the data can be used for correctness debugging and performance monitoring.

The rest of this paper is organized as follows. Section 2 defines the goals of K42's tracing facility and the technology we used to achieve them. Section 3 describes our implementation using atomic primitives to avoid locking to log variable-length events and provide random access to the data stream. In Section 4 we describe how the tools used the data generated by infrastructure to help us tune K42's performance and to debug the system. This section demonstrates the importance of having very efficient trace events always in the system and available to be dynamically enabled. Section 5 discusses related and future work. Section 6 concludes.

2 Goals

The tracing infrastructure in K42 was designed to meet several goals. The combination of mechanisms and novel technology we employed allowed us to achieve the following:

1. Provide a unified set of events for correctness debugging, performance debugging, and performance monitoring.
2. Allow events to be gathered efficiently on a multiprocessor.
3. Allow efficient logging of events from applications, libraries, servers, and the kernel into a unified buffer with monotonically increasing timestamps.
4. Have the infrastructure always compiled into the system allowing data gathering to be dynamically enabled.
5. Separate the collection of events from their analysis.
6. Have minimal impact on the system when tracing is not enabled, and allow for zero impact by providing the ability to "compile out" events if desired.
7. Provide cheap and flexible collection of data for either small or large amounts of data per event.

In addition to a unified tracing infrastructure, there are other performance evaluation mechanisms that can be used to understand operating system and machine performance. These fall into two classes: 1) operating system counters/record keeping data, e.g., the number of page faults in a given process and 2) hardware counters, e.g., cache misses. Both of these mechanisms are outside the scope of the tracing facility, though trace events may be used to log information gathered by such counters and later analyzed.

By doing so, the trace infrastructure may be used to study memory bottlenecks, memory hot-spots, and other I/O interactions by logging hardware counter events. e.g. cache-line misses. Integrating the hardware counter mechanism and the tracing infrastructure allows the counters to be sampled and understood at various stages throughout the programs or operating systems execution yielding a better understanding of system behavior.

Lockless event logging for achieving goals 1-4 and 7

The ability to log with acquiring any locks, events helps achieve many of the goals we set for K42's tracing infrastructure. Because logging is cheap and does not require locks, it can be used for purposes ranging from performance monitoring to correctness debugging. The lockless nature of the logging is a key aspect to being able

to share a single unified buffer across kernel, application, and server space. Further, the lockless algorithm allows variable-length events to be logged, facilitating logging small or large events.

Using a single buffer and lockless event logging introduces potential integrity and security issues. The buffer integrity issues are discussed in detail in Section 3.1. Our algorithm guarantees users cannot cause the kernel to crash or perform an errant write. Users can, however, write over parts of the buffer containing kernel or other user data. Therefore, this scheme is not suitable for auditing applications. Security concerns of a user having access to another user's or kernel's data can be addressed by backing the trace buffer of each process with different underlying physical memory similar to the relayfs[18] channel scheme in Linux.

Flexible and unified events for achieving goals 1 and 5-7

Other operating systems have used different infrastructures to gather events for correctness debugging as opposed to performance debugging or monitoring the system. Some even have multiple ways to gather information for either or both of performance debugging and monitoring. There is a trade-off in determining the number and kind of tracing facilities to incorporate. If the tracing facility is tailored to the specific needs of what it will be used for, then it will more closely match requirements of that use (correctness debugging for example).

We contend, however, that a unified and flexible system can provide the same capabilities while yielding advantages. Multiple systems for gathering events have a couple of disadvantages. In places where an event is important to multiple systems, more than one event needs to be logged. Also, to log that event correctly at each trace point in the code, the programmer has to know which system the event is intended for. This places an unnecessary burden on the programmer and can lead to errors. In code with multiple trace systems, it is typical to find more than one event logged at places throughout the code causing additional negative performance implications. Multiple trace logs complicate post-processing tools. Designing one flexible efficient system provides better performance, is simpler to code and simpler to update, and yields more understandable trace logs.

With the unified K42 tracing infrastructure, the programmer logs all important events to a single trace buffer, and separately, analysis tools using the data can decide which events to display for a given purpose. Having a unified facility allows a single check to be applied over all the tracing events to determine whether to trace or not, thereby reducing the impact on the system when tracing is inactive. A single facility also provides a more convenient mechanism by which to compile out all events.

One of the key advantages to a unified facility is illustrated by an example from our experience. In a particular performance debugging session, we were observing long lock hold times from our lock contention analysis (see Section 4.6). Because we had integrated scheduling events (in some systems these would be different mechanisms), we were able to see that there were context switches between the lock acquire and release events allowing us to understand what was actually occurring to cause the unexpected long hold times.

User-mapped per-processor buffers and control structures for achieving goals 2 and 3

In K42, good multiprocessor tracing performance is achieved by storing all the frequently referenced tracing data structures in memory bound to a specific processor. This allows all accesses to trace structures on separate processors to be independent, thereby yielding good scalability. To allow fast logging of events from user space, these control structures, containing for example the current index, and the trace buffers themselves, are mapped into each application's address space. On hardware architectures allowing it (e.g. PowerPC, MIPS), timestamps are obtained via a cheap user-level interface.

A process may be migrated in the middle of logging a trace event, potentially garbling the buffer on one or both of the processors involved in the migration. Though we have a couple of solutions (disabling migration, or just ensuring all data is written to the old buffer) we have not implemented them in K42 because the probability of this occurring is reduced by K42's emphasis on locality, K42's Linux emulation layer[1], and the fact that kernel threads do not migrate.

Variable-length events for achieving goals 1 and 7

Each event logged in K42 can be of a different size. There are trade-offs between using fixed-length or variable-length events. Fixed-size events allow for simpler logging and reading out as the consumer of events always knows the starting point of an event. This allows validity bits to be used, and allows invalid events to be skipped. Fixed-length events allow easy random access to the data stream, aiding reading and displaying large trace files. The disadvantages of fixed-length events are that they waste space, they take longer to write (to disk or network) because extra data needs to be written for short events, and they make it complicated to log data that is larger than the fixed size. K42 obtains the benefits of variable-length events while retaining random accessibility. It does so by ensuring that events never cross medium-scale alignment boundaries. We insert filler events as necessary to align the event

stream. Trace analysis tools can skip to any of the alignment points in a large trace. This technique provides the advantages of variable-length events and still allows fast access to all parts of a large trace (more details in Section 3.2).

Major and Minor IDs and a single word trace mask for achieving goals 4-6

Trace events are assigned major and minor IDs. By limiting the number of major classes to 64, a single comparison of a major class bit against a trace mask variable can determine whether an event should be logged. The major ID is a constant value, and because the trace mask variable is frequently referenced it remains “hot” and no cache misses are incurred. This provides an inexpensive method for determining whether to log an event, allowing the infrastructure to always be compiled in. As described in Section 4, in K42 we left the tracing infrastructure in, but inactive, even when gathering benchmarking results.

3 Implementation

There are several key aspects to achieving efficient variable-length event logging from kernel and user space on a multiprocessor. In this section we describe the lockless logging algorithm, our variable-length event strategy, and some details of K42’s tracing infrastructure; more details can be found in Auslander et. al.[3]

3.1 Lockless Event Logging

Previous lockless logging schemes[15] used fixed-length events with valid bits. As described in Section 2 there are several advantages to using variable-length events (assuming the random access problem is solved). In K42 we designed and implemented an algorithm to allow us to log variable-length events without locking.

Conceptually, each process attempts to *reserve* enough space in the buffer immediately after the current index for the event it intends to log. Once the process makes a successful reservation, it may proceed to log its data. To reserve space, a process attempts to atomically increment the current index using a `compare and store`. The process that successfully increments (as determined by the return value of the `compare and store` operation) the index has the right to proceed to log data into the buffer. Failing processes retry. Figure 1 shows on the left, in step 1, two processes, A and B, attempting to log events of different lengths after the current index from the initial configuration in step 0. Each process attempts to atomically increment the current index by the size of the event being logged. The winner, in this case process B, will log the event immediately following the old current index (see step 2). This will be followed by process’s A data, assuming no other competing processes attempt to log more data (see step 3). Because it is important to guarantee monotonically increasing timestamps, processes must re-determine the timestamp during each attempt to atomically increment the index. If the timestamp was not determined as part of the atomic reserve operation then that process may be interrupted by another process execute this code and get the next slot in the buffer, but obtains an earlier timestamp.

The memory for logging trace events is logically divided into buffers. The size of this buffer in K42 determines the alignment boundary mentioned earlier. Once a buffer is full, the logging facility proceeds to the subsequent buffer, and the previous buffer is available to be written out (to disk or network). The pseudo code appears in Figure 2 and complete open-source C code can be obtained by downloading K42 at http://www.research.ibm.com/K42.

Although the lockless scheme has good performance, there are potential complications that arise from using the algorithm. A process’s execution may be interrupted after it has reserved space to log an event, but before it actually performs the log. The interruption can occur because the process is preempted, blocks for a long time, or is killed. Depending on when (where in the sequence of code in Figure 2) the process is interrupted, different problems occur. If the process has had a chance to write the trace event header, but not the data, then only the data will be unrecoverable. If, however, the process has not yet logged the event header, then it is possible the rest of the buffer will be uninterpretable (our tools have ways of handling this situation). Only by locking, making the kernel perform the log, and disabling interrupts can this problem be prevented (there are low-level kernel events, in kernel code that runs disabled, that would still exhibit the problem, for example tracing NMIs - Non-Maskable Interrupts). There are a set of possible methods to ameliorate the difficulties.

If the process’s execution was interrupted due to preemption, then it is likely the process will run again soon and finish filling in the event before another entity notices, thus posing no real problem. If the process was killed, then the data will not finish being logged. The last line of pseudo-code detects this situation. The `traceCommit` function updates a per-buffer count of the amount of data that has been logged to that buffer. The count is zeroed during the `start new buffer` code. When the code responsible for writing the data (to a network stream, file, etc.) writes this buffer, it can compare the amount of data logged to this buffer, with the buffer’s size and report an anomaly if they do not match. If the process was interrupted because of a long-blocking operation, it is possible that both the current buffer will not

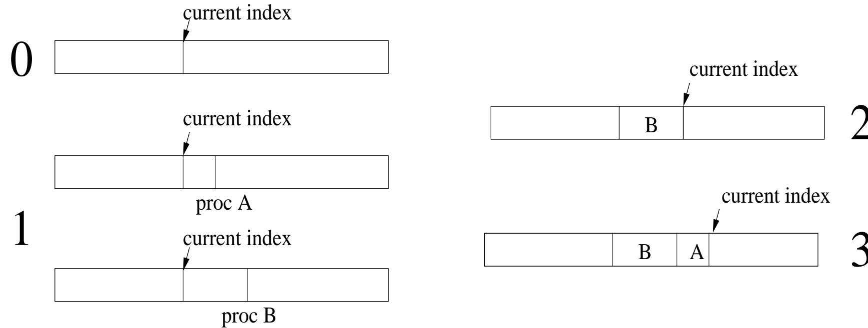

Figure 1: Illustration of Lockless Event Logging

```
traceReserve(length, *indexPtr, *timestampPtr)
    integer: oldIndex, newIndex
    TrcCtl: *trcCtlPtr
        update trcCtlPtr
        do
            oldIndex = trcCtlPtr->index
            newIndex = oldIndex + length
            if (newIndex >= buffer end)
                traceReserveSlow(length, indexPtr, timestampPtr)
                  // generates filler event, sets timestamp, moves to new buffer
                return
            *timestampPtr = getTimestamp()
        while (!CompareAndStore(&(trcCtlPtr->index), oldIndex, newIndex))
        *indexPtr = oldIndex & INDEXMASK  // confine index to buffer bounds

traceLog(majorID, minorID, data)
    integer: index, timestamp, length
        length = length of data + 1  // for header word
        traceReserve(length, &index, &timestamp)
        trcArray[index] = logTraceEvHeader(timestamp, length, majorID, minorID)
        trcArray[index+1 ... index+length-1] = data
        traceCommit(index, length) // optional, see explanation in text
```

Figure 2: Pseudo Code for Lockless Event Logging

have enough data logged, and that the same buffer, when reused in the future, will have too much (because the long-blocked process was unblocked and logged data into a recycled buffer). Again the per-buffer counts can detect this situation.

The chance of an error is small and highly dependent on the application mix that is run. For large scientific applications running one thread per processor, such errors will not occur. The probability increases on systems with a high degree of multiprogramming, i.e., those context switching between many applications. We have run entire benchmark suites without incurring any errors. With high probability (it is unlikely that random data will have the correct format of a trace event header) errors can be detected by the post-processing tools.

We have also considered other cheaper (than locking and disabling) mechanisms to eliminate such errors. In addition to a per-buffer count there are other possible ways to detect or minimize the occurrence of garbled data. For example, it is possible to set a flag in a process data structure and have the kernel avoid finishing killing a process if this flag is set. Other possibilities include cheaply zero-filling a buffer before use, or keeping a side array of valid bits for the header data.

In practice, the probability of garbling a buffer, and the ease with which tools can handle the situation, reduces its importance except perhaps in mission-critical code where obtaining every event is necessary.

3.2 Additional Tracing Infrastructure

Variable-length events with random access to data stream

As described in Section 2, variable-length events have advantages over fixed-length events. They allow more efficient writing of data, and they allow large events to be easily and efficiently logged. However, they interfere with the ability to randomly access the data stream. Tracing

files can become large; gigabytes per processor is common. Post-processing tools should not be forced to scan through the entire file when trying to display, for example, a middle 5 seconds of a program's execution.

In K42 we allow both variable-length events and random access to the file by ensuring that events never cross medium-scale (much larger than event sizes, but much smaller than file sizes) alignment boundaries, e.g. 128KB boundaries. We insert filler events as necessary to align the event stream at these points. Trace analysis tools can skip to any of the alignment points in a large trace and can begin interpreting events from that point. A filler event is just a header with a length equal to the remainder of the current buffer; no data need be logged. We have found empirically that 30 to 40 percent of events end exactly on a buffer boundary and because there are very few events larger than 4 64-bit words, this alignment in practice wastes very little space.

This technique does not provide completely random access, but is a close enough approximation that it allows post-processing tools to make it appear to the end user that the stream is completely random access.

Details of the Implementation

In this section, we present enough details of the tracing infrastructure to allow an understanding of the different uses of the facility presented next. More detail is available in our Performance Monitoring white paper[3], and in the code itself.

Trace events are assigned major and minor IDs, with a maximum of 64 major IDs. Trace log statements resolve to in-line functions that check a 64-bit trace mask to determine if tracing of a particular major ID is currently enabled. Thus, no functions calls are made in the case tracing is disabled. Further, the major ID is a constant, and the frequently used trace mask remains hot, so in practice no cache misses occur.

A trace event is broken up into a series of 64-bit words. The first word contains 32 bits of timestamp, 10 bits indicating the length, 6 bits for the major ID, and 16 bits of major-class-defined data, typically a minor ID. Following the first word are 0 or more 64-bit data words. We chose to log only 64-bit words because on some architectures smaller loads can be expensive, and because the vast majority of data being logged are 64-bit values or addresses. Macros provided with the tracing facility will pack multiple smaller quantities in one 64-bit tracing word, if needed.

The major ID classification also provides ease of adding additional events. Major classes are associated with subsystems within the operating system, `traceMem` for the memory subsystem, `traceProc`, and `traceIO`, etc. When an additional event is added, only files in the affected subsystem need to be recompiled. Per-major-ID macros allow events with a constant number of data words to be logged efficiently, without the use of variable argument functions. Events with non-constant-length data are logged using a generic function per major ID.

Efficiency of the Implementation

In addition to designing per-processor buffers to achieve a scalable multiprocessor implementation, we have carefully constructed the logging of each individual event to be as optimized as possible. Further, to achieve our goal of always leaving the tracing statements in the code, the mask provides a cheap way to determine whether to log an event or not.

The cost of checking the trace mask is 4 machine instructions. When running, and even when benchmarking K42, (for example, see the SPEC SDET graph in section 4) we leave the trace statements in. The overall performance degradation is less than 1 percent.

The majority of trace events are logged from C code. Logging a trace event in C takes between 70 and 80 PowerPC instructions. A 1-word 64-bit event requires 91 cycles (100 ns on a 1GHZ processor) with 11 cycles for each additional 64-bit word logged. We have not yet implemented optimized assembler code for generating trace events, except on critical code paths that are already coded in assembler. On these hand-optimized paths, logging a trace event requires about 30 instructions, including 6 instructions to update the per-buffer count.

4 Uses and Lessons

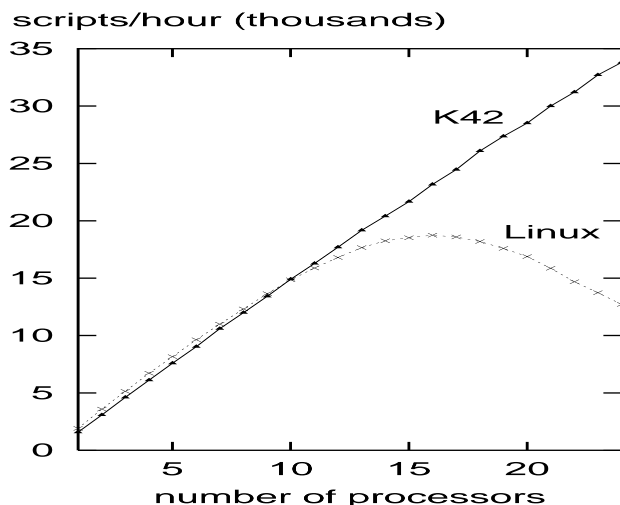

Figure 3: Results of running SDET on K42 and Linux on a 24-way multiprocessor

The tracing facility has been invaluable in helping us to

achieve good scalability. Some of the important features that have allowed the facility to be so effective are its low perturbation when enabled, allowing us to trace information like lock contention (see Section 4.6) or a fine-grain breakdown of the costs of different system calls (see Section 4.7). In addition, developers have used its features for correctness debugging (see Section 4.2). We have developed graphical tools to display the data as well (see Section 4.3). Some of the technology from K42's tracing infrastructure has been incorporated into Linux (see Section 4.1).

In this section we describe how we have used the facility to allow us to improve scalability in K42. The techniques we applied are applicable to supercomputing systems in general. The use of per-processor buffers, non-locking atomic event logging, cheap user-space logging, a single unified structure, etc., are techniques that could be applied in systems other than K42 and Linux.

We begin the section by describing our experience with the tools on a specific application. Figure 3 is a graph showing our experiment based on the SPEC SDET (Standard Performance Evaluation Corporation's Software Development Environment Throughput) benchmark. Briefly, this experiment runs a series of independent scripts that simulate a typical Unix time-shared environment by running commands such as `awk`, `grep`, and `nroff`. More details of the experiment may be found in Appavoo et. al.[1]. As evidence of the low impact of the tracing infrastructure, the data for the K42 graph shown in this figure was taken with the trace infrastructure compiled in.

When we made our first measurements of this experiment, K42's performance did not scale well. Though K42 was designed to scale near perfectly, quick or incomplete implementations of different code paths led to poor scaling. Further, our uniprocessor numbers were worse than Linux's. The challenge of achieving the performance illustrated in Figure 3 required utilizing the data tracing infrastructure and several of the tools described throughout this section.

The graphics tool helped us discover several performance problems. With it we noticed large idle periods on many processors when the benchmark started. These idle periods were clearly visible using the graphics visualizer but would have been difficult to discover via other methods. The excessive idle periods were caused by poor coordination between the timing and start routines of the benchmark. Another helpful feature of the graphics tool was the its ability to display throughout the benchmark's execution the points at which particular events occurred, thus allowing a sense of the benchmark's behavior to be achieved rapidly. In a similar manner, we used the graphics visualizer to help us determine what happened when our lock behavior degraded abruptly.

It was the lock analysis tool, which allowed us to understand and pinpoint the bottleneck locks, that proved to be most valuable in helping us achieve scalability. We went through a series of iterations where we used the lock analysis tool to determine the most contended lock in the system, fixed it, and then ran the tool again to identify the next most contended lock. We performed this operation until there were no more seriously contended locks.

We also used tracing to study our uniprocessor performance using a tool that produces a fine-grained analysis of elapsed time. A breakdown of page faults indicated that we needed to improve fork performance, which we accomplished by replicating state lazily in the child after a fork. The tool also allowed us to understand whether the behavior degradation (with respect to Linux) was coming from the user code, our Linux emulation code, or our kernel code. Throughout the process, the single tracing infrastructure was able to provide the data needed by the various tools described in this section.

It is interesting to note that there was some resistance early in the effort by core kernel developers (much the same as in the Linux kernel development community and previous operating system development efforts) as they did not perceive the usefulness of these tools in understanding system behavior. The usefulness of the tools however was proved over time as they allowed us to quickly and effectively tune the system. Many of the core kernel developers had used some tools on previous operating systems projects including Hurricane, IRIX, AIX, and Linux, but the efficiency, pervasive subsystem coverage, and ease of use of the infrastructure allowed quicker and more effective performance optimizations. In the rest of this section we examine the various tools that utilize the data provided by the tracing infrastructure.

4.1 LTT and relayfs

The Linux Trace Toolkit (LTT)[17][16] is the most-used tracing facility in Linux. Although LTT is not currently part of the kernel, it is included in many of the popular distributions, including UnitedLinux, MontaVista, Lineo, Debian, ELinos, and Denx, and efforts are underway to have it included it in the kernel. In the last year, several aspects of the technology described above have been integrated into LTT with positive results. More recently, relayfs[18], a mechanism for transferring data from kernel to user space in Linux, has also incorporated aspects of K42's tracing technology.

An order of magnitude performance improvement was achieved when this technology was applied to Linux. The three primary aspects providing this performance improvement were the lockless logging of events, per-processor buffers, and more efficient timestamp acquisition. Cheap user-mapped buffers are currently under development.

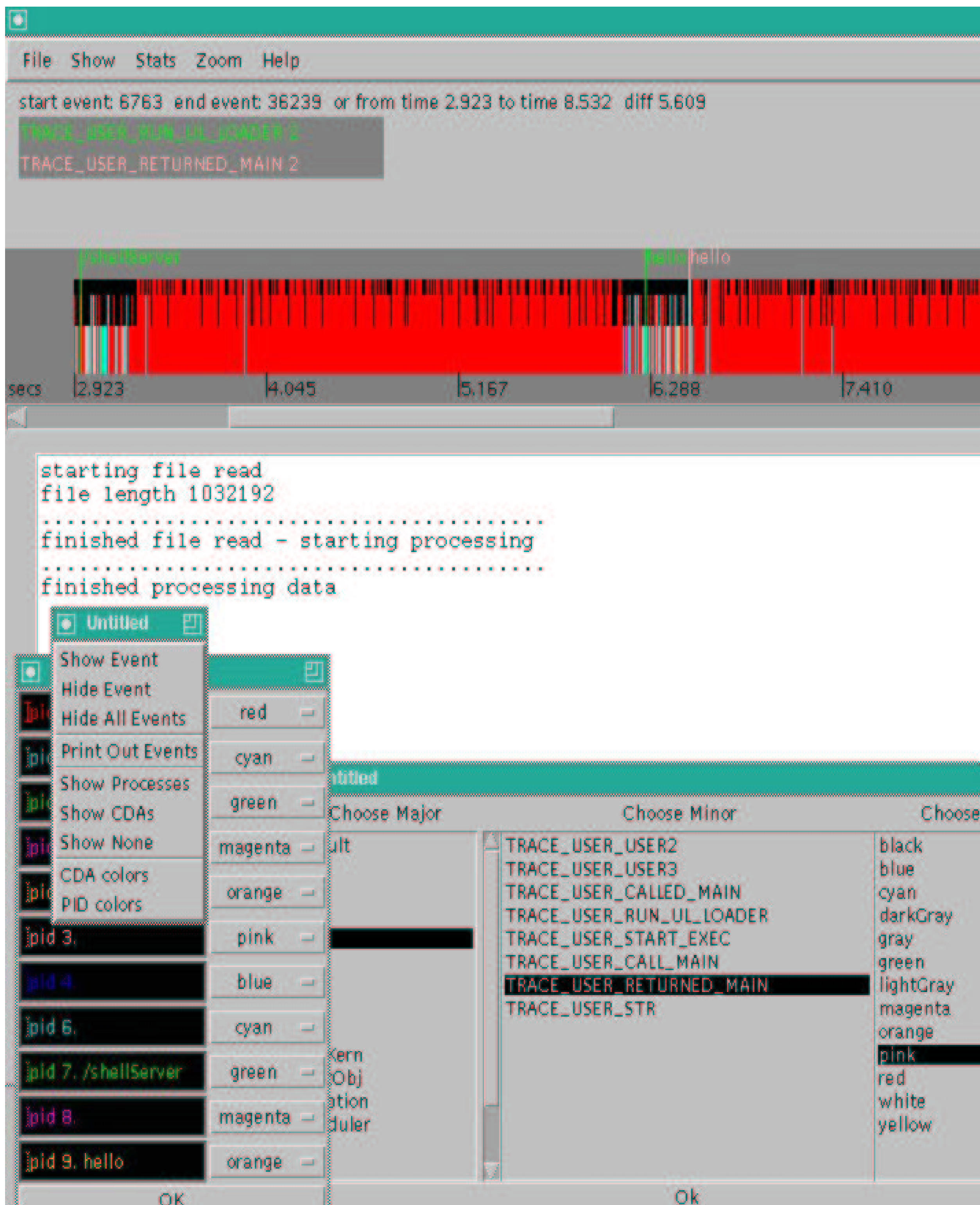

Figure 4: Graphical Viewing Tool

The default option in LTT is now to use the lockless logging though the locking option is still available. The locking option, which disables interrupts and process-state transitions, though slower, provides a greater likelihood that events will not be garbled. There are however, still cases when the locking scheme may lose events. The LTT mechanism has been extended to use a separate buffer per processor. On PowerPC hardware platforms (where K42 was developed) a cheap synchronized clock is available. However, x86 architectures do not provide such a clock. Instead, LTT logs the cheaply available `tsc` with each event, and only at the beginning and end is the more expensive `get_time_of_day` call made allowing synchronization between different processors' buffers through interpolation of the `tsc` values between the `get_time_of_day` values.

4.2 Correctness Debugging

In addition to performance tuning, the tracing facility was designed to be used by kernel and applications developers to help correctness debug the system. A feature of the design aiding this effort is management of the trace array for each processor as a circular buffer. When the buffer becomes full new events overwrite old events so that if the kernel should crash, the most recent activity recorded by the tracing infrastructure is available. This "flight recorder" functionality can be accessed from the debugger via a function call that prints out the last set of trace events. It has features to show only certain type of events and has control as to how many events it displays. If the kernel is not stable enough to call this function, a crash dump tool can access the trace log providing similar functionality. We have not implemented the crash dump tool yet.

System developers have used the trace facility to correctness debug the system. For example, a deadlock in the file system space was tracked down with the tracing facility. To discover the deadlock, it was important to track the order of all the different requests being received from the file system's numerous clients. A `printf` solution would both have been too clumsy and would have changed the timing thereby masking the deadlock. Instead, a trace file was produced and post-processed to detect where the cycle had occurred.

Because of the ease of adding additional events and the efficiency with which they are logged, other developers have used the tracing facility to obtain statistics about the relative frequency of different paths taken through code. A typical alternative solution would have been to design a one-off counter solution that would have been removed once the information was gathered. Because the tracing facility was cheap and easy enough to use, it made this process simpler.

4.3 Graphical Analysis

It is hard to understate the importance of a graphical tool in understanding the data. Certain behavior does not become evident unless the data is visualized. We have argued many times with kernel developers who claim not to need any such capability, but inevitably the graphic and/or performance analysis tools turn up a previously unsuspected problem and the developer becomes a convert. This has been true in both the current and past operating systems we have worked on. The graphical tool we have developed has focused on helping us understand operating system behavior. In addition, we are working to convert K42's trace file to one the LTT visualizer understands. The data could certainly feed machine- or application-centric graphical tools.

The screen snapshot shown in Figure 4 shows the primary view of the kmon visualization tool. The timeline in the top middle provides a bird's eye view of the events occurring in the system. The timeline view provides the developer with a visual sense of what is occurring in the system and how active the system is. The user can zoom in or out to get a sense of the system behavior at different granularities.

Other aspects of the tool allow specific events to be marked and counted. In Figure 4, two such events have been selected. TRACE_USER_RUN_UL_LOADER (difficult to see in a black and white printing) and TRACE_USER_RETURNED_MAIN are events indicating initialization and termination of a process. The graphical tool, when the mouse is clicked in the timeline area, will produce a listing of every event that occurred around the time period the mouse was clicked in. A sample listing appears in Figure 5. There are considerably more features available with the tool.

The crucial feature the graphic tool provides is the ability to rapidly get a sense of system performance and to detect quickly any anomalous behavior even if it was not explicitly under consideration as a problem. As an example, one time when viewing a data file, 10ms continuous chunks of red (kernel time) were appearing. To any tool that provide summaries or averages this would have been in the noise because there were not that many of them. However, they visually stood out. Once we observed the behavior we were able to trace it down to an unexpected poll condition and fix the behavior.

4.4 Listing of Every Event

We have a tool that takes a binary trace file and produces the textual output shown in Figure 5 (left column is time in seconds). The event names in the second column and the event description in the third column, are generated from an `eventParse` structure filled in when a developer de-

```
21.4747350 TRC_USER_RUN_UL_LOADER       process 6 created new process with id 7 name /shellServe
21.4747422 TRC_EXCEPTION_PGFLT          PGFLT, kernel thread 80000000c12b0f90, faultAddr 405e628,
21.4747882 TRC_EXCEPTION_PGFLT_DONE     PGFLT DONE, kernel thread 80000000c12b0f90, faultAddr 405
21.4748091 TRC_EXCEPTION_PPC_CALL       PPC CALL, commID 0
21.4748530 TRC_MEM_FCMCOM_ATCH_REG      Region 800000001022cc98 attached to FCM e100000000003f30
21.4748709 TRC_MEM_FCMCRW_CREATE        TRC_MEM_FCMCRW_CREATE ref e100000000003f90
21.4749142 TRC_EXCEPTION_PPC_RETURN     PPC RETURN, commID 600000000
21.4749247 TRC_EXCEPTION_PPC_CALL       PPC CALL, commID 0
21.4749573 TRC_MEM_REG_CREATE_FIX       Region default 10000000 created fixlen addr 113000
21.4749773 TRC_MEM_REG_DEF_INITFIXED    region default init fixed 80000000102b7c00 addr 10000000
21.4749873 TRC_MEM_ALLOC_REG_HOLD       alloc region holder addr 10000000 size 113000
21.4749962 TRC_MEM_ALLOC_REG_HOLD       alloc region holder addr 10000000 size 113000
21.4750293 TRC_MEM_FCMCOM_ATCH_REG      Region e100000000003fa0 attached to FCM e100000000003f90
```

Figure 5: Trace Event Listing from K42

```
histogram for pid 0x1 mapped filename servers/baseServers/baseServers.dbg
  count   method
    904   FairBLock::_acquire()
    585   HashSNBBase<AllocGlobal, 0l, 8l>::add(unsigned long, unsigned lon
    386   DispatcherDefault_IPCCallEntry
    265   MemDesc::alloc(DataChunk*, unsigned long, unsigned long&, unsigne
    254   HashSimpleBase<AllocGlobal, 0l>::find(unsigned long, unsigned lon
    227   _wordcopy_fwd_aligned
    159   XHandleTrans::alloc(Obj**, BaseProcess**, unsigned long, unsigned
    141   TmpRWLock<BLock>::releaseR()
    135   DentryListHash::lookupPtr(char*, unsigned long, NameHolderInfo*&)
    134   HashSimpleBase<AllocGlobal, 0l>::extendHash()
    130   DirLinuxFS::externalLookupDirectory(char*, unsigned long, DirLinu
```

Figure 6: Breakdown by Time Within a Single Process

fines a new event. This structure contains a description of the data in the event (i.e., binary data, string, and the size of the data), and a printf-like formatted string describing how to print it.

Complete details of the self-describing string can be found in the source. Briefly, the structure contains 3 fields. The first field is a macro `__TR(arg)` that allows `arg` to be used as both a constant and string by the tools. The second field consists of a string that defines the format of the binary data in the trace event. It has as many space-separated tokens as there are values in the event. The tokens can be `8`, `16`, `32`, `64`, or `str`, indicating 8, 16, 32, 64 bits of data or a string. The third field is a printf-like string indicating how the data should be printed out. The tokens from the field are numbered starting at 0 and may be referenced in the third structure by %N[*format*], where the %N indicates the numbered token from the second field (the numbers do not need to be in order in the third field) and the *format* is a printf-like string indicating how to print the value. An example from our source code follows:

```
{__TR(TRACE_MEM_FCMCOM_ATCH_REG), "64 64",
"Region %0[%llx] attach to FCM %1[%llx]"},
```

The structure allows tools to display events without any special knowledge of the events themselves.

4.5 Breakdown of Time by Process

An event that logs the program counter at random times is used to drive statistical execution profiling. Post-processing analysis maps the pc values to C function names and provides a sorted histogram of the routines that were statistically most active. Figure 6 shows a lock routine leading the list. We have trace events on the locking paths to help determine which locks account for this time as described in the next section.

4.6 Lock Contention

A particularly important performance aspect of large multiprocessors is lock behavior. One of the powerful techniques we have been able to use because of the efficient tracing facility is the ability to trace contended lock paths. The lock analysis tool has played a crucial role in helping us detect when a particular lock is generating contention, and how much that contention is affecting performance. Figure 7 shows the data the lock analysis tool produces. The left column is the total amount of time (over the given run) that was spent waiting for that particular lock. The next column is the number of times that lock was contended. The spin column is the number of times we have gone around the spin loop waiting for the lock. Spin count can be significantly different from time if the process fails

```
  top 10 contended locks by time - for full list see traceLockStatsTime
       time   count     spin     max time  pid
                                          call chain
3.466320753    1209 188795433  0.012220087  0x1
                                          AllocRegionManager::alloc(unsigned
                                          PMallocDefault::pMalloc(unsigned
                                          GMalloc::gMalloc()
0.684612632     573 37233770  0.007647854  0x0
                                          AllocRegionManager::alloc(unsigned
                                          PMallocDefault::pMalloc(unsigned
                                          GMalloc::gMalloc()
0.104643241   11885  4910595  0.000322320  0x1
                                          PageAllocatorDefault::deallocPages(unsig
                                          PageAllocatorUser::deallocPages(unsigned
                                          AllocPool::largeFree(void*,
0.075784944    8772  3526318  0.000471144  0x1
                                          PageAllocatorDefault::allocPages(unsigne
                                          PageAllocatorUser::allocPages(unsigned
                                          AllocPool::largeAlloc(unsigned
```

Figure 7: Lock Contention Analysis

to acquire the lock and blocks to wait for it. The next column is the maximum time a process ever waited to acquire this lock. The tool will sort on any of these columns. The next column indicates the PID the lock was associated with (PID 0 in K42 is the kernel and 1 is baseServers[1]). The final column is the call chain that led to the lock acquisition.

4.7 Fine-Grained System Behavior

K42 tracing data is detailed and fine-grained enough to allow us to attribute time accurately among processes, thread switches, IPC (inter-process communication) activity, page-faults, and transitions to and from the Linux emulation layer in user space. Further, in each such category, we can identify and track the page-faults that occurred, and the IPC calls that were made. Within server processes and the kernel we identify how much time is spent servicing IPC calls made by other applications, which is then categorized by function.

Figure 8 shows the detail we are able to achieve with the tracing data. In this paper we do not fully describe the data, however, it is clear that with the efficient tracing infrastructure significant amount of detail can be obtained and thus used to understand system behavior. In the following data, all times are in microseconds. The first column of numbers is the amount of time spent computing in that code, followed by the number of times it was called, followed by the number of events that occurred. For example, `SCexecve` accumulated 209.59 usecs, was called once, and contained 86 events. The second column represents the time and number of page faults that occurred because of these calls. For example, `SCexecve` incurred 15 page faults taking 273.20 usecs. The last column represents the same data for IPCs. For example, `SCexecve` made 34 IPCs with a total time of 691.53 usecs. Other relevant data is the "Ex-process" row, which is the number and time spent on calls for this process but outside of it (kernel and server time). Finally, at the bottom, is a list of thread entry points containing the number of times they were called and the amount of time they spent servicing requests.

5 Related and Future Work

Previous work for tracing operating systems such as AIX[5], IRIX[15], or Linux[17] have had limitations including using fixed-length events, only allowing tracing via system calls, requiring locking to log events, and using inefficient timestamp acquisition. The work described here addresses these problems and provides an implementation that efficiently logs variable-length events, does not require a system call to log an event, uses per-processor buffers, acquires timestamps efficiently, and provides lockless logging of events.

There has been considerable work in understanding parallel system behavior[14][13][7][9]. A large focus of this and other similar work[6][8][12] has been to reduce the amount of tracing information either through statistical sampling, dynamic insertion into code for areas of current interest, or choosing a few important events.

By choosing a few key events in Choices[7] the overall system behavior was able to be roughly understood. Other work such as Dyninst[6] and Paradyn[8] has examined how to dynamically insert events but was more targeted towards application space. But even KernInst[12], which is targeted at kernel instrumentation, has higher overheads

```
pid: 3d parent: 30 lpid: 163 lparent: 157
Exec:./runtest.sh /bin/rmdir
 SCbrk      :      8.39/4/8       f:                 p:    31.16/2
 SCchild    :    338.43/4/120     f:  1041.17/80     p:   107.45/18
 SCclose    :     26.07/2/13      f:    45.42/3      p:    42.49/4
 SCexecve   :    209.59/1/86      f:   273.20/15     p:   691.53/34
 SCexit     :     13.43/1/9       f:                 p:    24.19/5
 SCfstat    :      4.78/1/3       f:                 p:     6.69/1
 SCgetpid   :      1.01/1/1       f:                 p:
 SCmmap     :     53.39/4/42      f:                 p:   199.94/19
 SCmprotect :      0.56/1/1       f:                 p:
 SCopen     :     38.10/3/12      f:                 p:    60.60/3
 SCread     :     24.03/1/9       f:    42.04/2      p:    46.78/3
 SCrmdir    :     13.61/1/3       f:                 p:    53.92/1
 SCsigaction:     10.29/6/6       f:                 p:
 SCsigprocma:      4.54/1/2       f:    10.07/1      p:
 async      :      2.69/2/2       f:                 p:
 dispatcher :     32.71/1/13      f:    87.53/7      p:     9.77/3
 user       :   1718.56/27/104    f:  1304.87/76     p:
0 In-process total:  2500.18/434
----------------------------------------------------------------------------
 cleanup    :    929.41/1/5       f:                 p:
 fault      :   2804.31/184/186   f:                 p:
 ppc        :   1274.52/93/210    f:                 p:
0 Ex-process total:  5008.23/401
wall 10800.11/0
------------------------------------------------------------------------------
  CRT::ForkChildPhase2                                 255.32/2
  DispatcherDefault::AsyncMsgHandler                     4.05/3
  CRT::ForkWorker                                      246.10/4
  COSMgrObject::CleanupDaemon                          185.61/2
  MPMsgMgrEnabled::ProcessMsgList                        3.56/1
```

Figure 8: Fine-Grained Behavior Breakdown

than the facility described here. This overhead is due in part to the flexible and dynamic nature of KernInst requiring springboard and overwrite instructions. Recent work[13][14] has examined dynamic and adaptive instrumentation to focus collection on locally crucial aspects of performance.

Tracing and performance tuning multiprocessor operating systems has different constraints than understanding multiprocessor application performance. In operating systems there are a series of well known events that affect behavior, examples include context switch, I/O interrupt, IPC, etc. The ability to rapidly trace all these frequent events in the minimum time is key to understanding system behavior. For understanding operating system behavior, we believe a combination of highly optimized events in well-known locations will always be important. Building on that, tools like KernInst, or a similar Linux tool Dynamic Probes[4], will be used to complement the in-place tracing events. In fact, for this reason, DProbes is currently being integrated with LTT.

Even for operating systems, the importance of dynamic tools should not be overlooked. Dynamic tools are necessary when attempting to start monitoring in unanticipated ways an already installed and running machine. We are investigating using our hot swapping mechanism[10] to provide this capability in K42. Though the importance of being able to add dynamic events in the field is great, we believe that for kernel developers tuning their own code, the ease with which trace events can be added and the efficiency with which they operate will continue to make them the mode of choice. Thus, the importance of having an easy and efficient facility as described here is high.

While we have not focused on the end tools and visualization aspect, it is the most important aspect in the final understanding of system behavior. Almost all of the above cited work included a substantial attempt to cull the pertinent information into a crisp graphical representation. There have been books[11] written on how to view and analyze system performance data. Our tools have sufficed to meet our current needs, but integrating them with more powerful techniques remains an important aspect of our continuing performance monitoring work.

Future work

The tracing infrastructure has provided tremendous benefit to us in understanding the behavior of K42 and achieving good scalability. K42 is becoming a stable platform on which other researchers may experiment. As more focus is placed on middleware and applications, the analysis tools will need to continue to be developed to provide more application-centric information. The base infrastructure used to understand the system should provide sufficient capability to gather any data needed by these tools.

An immediate area of future work is converting the output stream produced by K42's trace facility so that it can be read by LTT's visual display toolkit[16]. That package provides a nice model to understand thread interactions. Another area we have not yet focused on, that will be more important as our application base grows, is providing protection and security between different applications. Currently, all data is logged to a single shared buffer. Although this has good performance and analytical properties, different users may not desire to have information about their behavior available to other users. To solve this, we intend to map in different buffers to user applications that do not have sufficient privileges to see all data. To date, we have focused on using the infrastructure to understand the behavior of the kernel and on helping to correctness debug the system. The infrastructure was designed to facilitate dynamic tuning of the operating system. We are investigating how to integrate our hot-swapping[10] infrastructure with the tracing infrastructure in order to provide feedback for the system to tune itself.

6 Conclusions

We have developed an efficient, unified, and scalable tracing infrastructure that allows for correctness debugging, performance debugging, and performance monitoring of a large multiprocessor operating system. Some of the key features of the tracing infrastructure include lockless logging, per-processor buffers accessible from user and kernel space, variable-length events, and a randomly accessible data stream.

The tracing facility is well integrated into our code, easily usable and extendable, and efficient, providing little perturbation of the running system. We described different ways to analyze the tracing data including lock contention analysis and graphical display. The facility has been invaluable in helping us achieve good scalability in K42.

K42 is under active development at IBM Watson, and collaborating universities. Interested parties may learn more about the project at the following web site: http://www.research.ibm.com/K42.

Acknowledgments

An operating systems project is a huge undertaking and being able to explore performance monitoring and tracing infrastructure in a scalable operating system environment has provided a unique opportunity to develop innovative technology. Thanks to the whole K42 team at Watson, as well as our university collaborators. Thank you to Dilma da Silva for her careful reading and useful suggestions. K42 members, including the authors, are: Jonathan Appavoo, Marc Auslander, Dilma da Silva, David Edelsohn, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert Wisniewski, and Jimi Xenidis.

References

[1] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Freenix*, pages 323–336, San Antonio, TX, June 9-14 2003.

[2] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. *K42 Overview*. IBM Research, http://www.research.ibm.com/K42, August 2002.

[3] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. *K42's Performance Monitoring and Tracing*. IBM Research, http://www.research.ibm.com/K42, August 2002.

[4] IBM Linux Technology Center. Dynamic probes. http://www-124.ibm.com/developerworks/oss/linux/projects/dprobes/.

[5] IBM Corporation. Aix version 3.1 for risc system/6000 performance monitoring and tuning guide. Technical Report SC23-2365-00, IBM Corporation.

[6] Dyninst. An application program interface (api) for runtime code generation. http://www.dyninst.org/.

[7] D. Kohr, X. Zhang, M. Rahman, and D. Reed. A performance study of an object-oriented parallel operating system. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, November 27 2000.

[8] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.

[9] Daniel A. Reed, James Arendt, Ruth Aydt, Thomas Birkett, David Jensen, Tara Madhyastha, Bobby Nazief, Ted Nelson, Robert Olson, and Brian Totty. Scalable performance environments for parallel systems. In *Sixth Distributed Memory Computing Conference*, pages 562–569, Portland OR, April-May 1991.

[10] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX*, pages 141–154, San Antonio, TX, June 9-14 2003.

[11] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization*, volume 1, chapter 20 Visualization of Dynamics in Real World Software Systems, Doug Kimelman, Bryan Rosenburg, and Tova Roth, pages 293–314. MIT Press, 1998.

[12] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI 99: Third Symposium on Operating Systems Design and Implementation*, pages 117–130, New Orleans, February 1999.

[13] Christian Thiffault, Michael Voss, Steven T. Healey, and Seon Wook Kim. Dynamic instrumentation of large-scale mpi/openmp applications. In *IPDPS 2003: International Parallel and Distrubuted Processing Symposium*, page to appear, Nice France, April 2003.

[14] Jeffrey S. Vetter and Daniel A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *SC 99 Proceedings of SC 99*, page electronic publication, Portland OR, November 1999.

[15] Robert W. Wisniewski and Luis F. Stevens. A model and tools for supporting parallel real-time applications in unix environments. In *Proceedings of The 12th IEEE Real-Time Technology and Applications Symposium*, pages 126–133, Chicago Illinois, May 15-17 1995.

[16] Karim Yaghmour. Ltt web page. http://www.opersys.com/LTT/index.html.

[17] Karim Yaghmour. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[18] Tom Zanussi, Karim Yaghmour, Robert W. Wisniewski, Michel Dagenais, and Richard Moore. An efficient unified approach for trasmitting data from kernel to user space. In *OLS 2003 - Ottawa Linux Symposium*, page to appear, July 23-26 2003.