SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1(1), 1–21(DECEMBER 1994)

Kitrace: Precise Interactive Measurement of Operating Systems Kernels

GEOFFREY H. KUENNING
Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90024 USA
E-mail: geoff@ficus.cs.ucla.edu

SUMMARY

`Kitrace` is a software tool that allows dynamic interactive measurement of UNIX kernel performance to much greater precision than that available from kernel profiling. Developers can measure, to microsecond resolution, the time required by a complex kernel activity, including time spent waiting for I/O activity or user processes. `Kitrace` has also proven useful for debugging, especially in situations where traditional breakpointing would be undesirable or would change the behavior of the kernel.

KEY WORDS Operating systems Performance measurement Tracing Debugging

INTRODUCTION

Traditionally, operating-systems performance measurement has been a difficult and indirect task. Profilers such as `gprof`,[1] while simple to implement and widely available, suffer from statistical inaccuracies[2] and cannot identify queueing or I/O delays. Object-code modifiers[3, 4] are clumsy to use and difficult to target precisely. Hardware analyzers,[5, 6] while powerful, are expensive and limited to special installations. One of the most flexible techniques is software-based trace collection,[7-15] but the inconvenience of inserting trace statements has hampered the usefulness of this method.

This is especially unfortunate because of the size and complexity of modern operating system kernels. Kernel performance and reliability have a direct impact on the performance and reliability of the overall system, so that the need for good tools is even greater than that of many user-level programs.

We have developed a new tool, `kitrace`,[*] which complements profiling and *ad hoc* measurement by allowing precise interactive measurements of running kernels. A user-level interface allows tracepoints to be set at (nearly) any kernel instruction. When a tracepoint is reached, precise timing information is captured in a trace buffer, together with a few variables selected by the user to aid in analysis. The user-level interface program prints the buffer and allows easy modification of tracepoints based on experimental results.

Although `kitrace` was not specifically designed as a debugging tool, it has also proven very useful for that purpose, especially when analyzing timing-dependent problems, because

[*] For *kernel interactive trace*, and pronounced *KIH-trace*.

0038–0644/94/010001–21$15.50 *Received 2 December 1993*
Revised 1 August 1994

of its minimal impact on kernel behavior.

INTERACTIVE KERNEL TRACING

`Kitrace` is a tool that falls between profiling and special-purpose code in the measurement spectrum. Like profiling, it provides generalized facilities that may be used on a running kernel at any time. Like special-purpose code, it allows precise and detailed measurements that are limited to the areas of interest. In addition, although it was not developed for that purpose, `kitrace` has proved to be a useful adjunct to traditional methods of kernel debugging.

Overview

The `kitrace` package is divided into a user-level control program and a small amount of kernel support code. When the user requests a tracepoint, the control program uses the UNIX memory-access device (`/dev/kmem`) to insert a trap instruction into the kernel at the appropriate point, and to record trace-control information in a dedicated structure.

When a tracepoint is reached, the kernel support code captures the program counter, the current time (to microsecond resolution, if the hardware supports it), and, if requested, a small amount of relevant data. It places all of this information into a trace buffer in kernel space, and resumes normal kernel execution by single-stepping past the trapped instruction, using a standard technique discussed below. Our performance measurements, presented later, show that the entire process takes only a few tens of microseconds on modern CPU's. The low cost of tracing means that measurement rarely has an impact on the software being examined, and the tracing cost can be quantified precisely enough to be taken into account in analysis.

After data has been collected, the user-level control program reads the trace buffer, again using `/dev/kmem`, and prints it in a standard format. The control program can also clear the trace buffer, delete or modify tracepoints, and perform high-level operations such as temporarily stopping all tracing. All operations on `/dev/kmem` are carefully ordered to avoid race conditions that might crash the kernel or cause trace data to be lost.

UNIX kernel internals

The examples in this paper are chosen from actual measurements made on the UNIX kernel running the Ficus operating system,[16, 17] which is a modified version of SunOS 4.1.1, on a Sparcstation IPC. Since `kitrace` is designed for kernel development, its use presupposes familiarity with the code being measured. For those readers unfamiliar with the aspects of the UNIX kernel used in our examples, a brief summary is in order.

System calls

Although any instruction in the kernel, with the exception of those involved in the tracing process itself, can be traced, for simplicity most of the examples trace the behavior of the `sync` system call, which causes dirty memory buffers to be written to disk and network file systems. This operation is useful for demonstrations because it can be invoked by any user and it causes both disk and network I/O. In the kernel, the first instruction of the system call is found at the label `_sync`. Some of the examples trace other instructions internal to the `_sync` routine. These instructions are discussed as part of the examples.

A few examples and measurements use other calls. The `open` and `close` calls perform the standard file operations associated with those verbs. They were chosen to illustrate how `kitrace` can be used to examine some types of user-process behavior. The `getpid` call allows a user process to discover its system-assigned process ID, or PID. Because this call simply returns an integer to the caller, it can be easily executed many times in succession to collect statistical performance data on the overhead introduced by `kitrace`.

Internal kernel routines

The examples also make use of several other routines which are available only internally to the kernel. Some of these are invoked by the `_sync` routine to flush buffers to various file systems. All of these routines have names of the form *xxx*`_sync`, where *xxx* is indicates a file-system type. Many filesystem types do not do any actual work. Important types include `spec`, which handles meta-data such as block-location information for the hard-disk filesystem, `ufs`, which handles file contents for the hard-disk filesystem, and `nfs`, which implements Sun's Network File System.[18]

Primary disk scheduling is carried out by the `_sdstrategy` subroutine, which is responsible for initiating I/O to disk drives and for ordering the I/O queue to optimize head movement. This routine may be called at any time, regardless of whether the disk is busy. If necessary, it must queue requests for future execution. However, if the queue becomes too long, it is permissible to suspend the caller until space becomes available.

All disk interrupts are handled by the routine `_sdintr`. Although there are rare instances when interrupts signal something other than disk completion, the examples in this paper assume that every entry to `_sdintr` represents completion of a disk I/O. The interrupt routine will notify and schedule any process that is waiting for I/O completion, and will initiate the next disk I/O if any requests are pending.

Features

The user-level interface to the package is a single program named `kitrace`. This program supports both a command-line and an interactive mode; all features are available in both modes. Examples in this paper will use the less-cryptic interactive syntax. The interactive commands are summarized in Table I.

Simple tracing

In its simplest form, `kitrace` can be used to set a tracepoint and later print the results. For example, we can use the `trace` command to set tracepoints at the kernel `sync` system call, wait a few seconds for some activity to take place, and use `print` to show the times of the observed operations. This is illustrated in Figure 1.* In this example, the tracepoint is named symbolically. Tracepoints can also be specified in hexadecimal, or as the sum of one or more symbolic or hexadecimal values. `Kitrace` always attempts to look up a non-numeric value symbolically first, and if possible reinterprets it as hexadecimal if the symbol cannot be found. Hexadecimal interpretation can always be forced by prepending the sequence `0x` to a value.

In the basic output, we see the names of the tracepoints reached (only one in this case), the times they were executed, and the elapsed time between successive traces. Thus, we can see

* In all examples in this paper, system activity was generated by another process after tracepoints were set and before the `print` command was issued. This activity generated the trace data displayed by the `print` command.

Table I. Interactive trace commands

`clear`	Clear the trace buffer
`print`	Print the trace buffer
`printclear`	Print and then clear the trace buffer
`follow`	Print and clear repeatedly
`freeze`	Freeze tracing
`unfreeze`	Unfreeze tracing
`list`	List currently-set tracepoints
`delete` *addresses*	Delete tracepoints
`trace` *address [options]…*	Set tracepoints
`status`	Report tracing status
`help`	Print a help message
`quit`	Exit kitrace

that three `sync` calls were made, at intervals ranging from about 12 to 20 seconds, while we were waiting to type the `print` command.

Data capture

Frequently, it is desirable to capture more than just the time of a tracepoint. The `trace` command supports a number of options, summarized in Table II, to allow data collection. For example, the `PID` option to the `trace` command causes `kitrace` to record the ID of the currently-running process when the trace is captured. Figure 2 shows how we can use the `PID` option to learn which UNIX processes are executing the `sync` calls.*

Note that previous entries in the trace buffer have not been removed. New entries are appended to the buffer until an explicit `clear` command is issued. It is not possible to add new information to old entries, so we still do not know the process ID's of the programs that did the first three `sync`s. But we can now see that five subsequent `sync`s were done by three different processes. In this case, we used the `ps` command to learn that process 216 is the `update` daemon that performs a `sync` approximately every 30 seconds.

More information can be captured than just the PID, although the need to keep the mechanism

* In this and some subsequent examples, the output has been edited to simplify the presentation. Actual `kitrace` output will differ slightly in appearance. Completely omitted lines are indicated by ellipses.

```
(kitrace) trace sync
(kitrace) print
_sync           Nov 10 10:31:36.565348 (            )
_sync           Nov 10 10:31:49.423993 (    12.858645)
_sync           Nov 10 10:32:10.157718 (    20.733725)
```

Figure 1. Simple tracing of the `sync` system call

Table II. Options to the `trace` command

`PSR`	Collect Program Status Register
`ALLREGS`	Collect all registers
`REG` *name*	Collect the named processor register
`CONST` *value*	Record a constant value
`STACK` *offset*	Collect a value from the stack
`FRAME` *offset*	Collect a value from the stack frame
`LOC` *address*	Collect the contents of a memory location
`PID`	Record the current process ID
`FREEZE`	Halt tracing after this tracepoint
`LATEFREEZE` *count*	Halt tracing after *count* more tracepoints
`UNFREEZE`	Restart tracing at this tracepoint
`ONLYPID` *pid*	Cause this entry to apply to only one process
`BASE`	Base time differences from this tracepoint
`ADJACENT`	Base time differences on adjacent tracepoints

lightweight limits the user to collecting at most a small number, currently 3, of items selected from the list in Table II. For example, on the machine used to generate these examples, the work of `sync` is actually done by making indirect calls to a number of routines listed in a table. Since the pointer to the called routine is held in register `g1`, we can discover the identities of these routines by using the `REG` option to capture this register, in both symbolic and hexadecimal form, at the point of the indirect call (`sync+50`), as shown in Figure 3. Here we see the complex process that `sync` initiates. We can also tell, based on the elapsed times, that most of the time is spent in `ufs_sync` (because of the long interval between the entry to it and the entry to the next routine), and that `nfs_sync` is the last routine that does any significant work. The elapsed times recorded for most of the remaining routines are so close to the overhead of a tracepoint, analyzed below, that we can conclude even without looking at the code that they must be stubs.

Besides the process ID and machine registers (including the processor status word), `kitrace` also allows the user to capture data from the stack, the current stack frame, or a fixed location in memory. There is currently no facility for following pointers, but the effect of pointer-following can usually be achieved by capturing a register at an appropriate point in the code being examined.

Control facilities

Sometimes it is necessary to set tracepoints in heavily-used routines such as `open`. In such cases, the trace buffer quickly fills with clutter, making it difficult to separate useful data from noise, and possibly causing traces to be lost due to buffer overruns. `Kitrace` provides several features to deal with this problem.

Per-process control. Suppose, for example, that we are interested in examining the behavior of the `open` and `close` system calls in various situations. It is easy to write a program to exercise the calls, but so many other programs also use them that it is difficult to separate out the relevant data or prevent trace buffer overruns. Figure 4 shows how we can use the

```
(kitrace) trace sync PID
kitrace:  Tracepoint at _sync already exists, updating
(kitrace) print
_sync           Nov 10 10:31:36.565348 (         )
_sync           Nov 10 10:31:49.423993 (12.858645)
_sync           Nov 10 10:32:10.157718 (20.733725)
_sync           Nov 10 10:32:19.577274 (09.419556) pid 7705
_sync           Nov 10 10:32:25.929029 (06.351755) pid 216
_sync           Nov 10 10:32:57.725656 (31.796627) pid 216
_sync           Nov 10 10:33:28.274735 (30.549079) pid 7735
_sync           Nov 10 10:33:30.226706 (01.951971) pid 216
```

Figure 2. Capturing process ID's

ONLYPID option to limit tracing to a selected process by giving its ID.

Here, we can see the process traced did a single open, followed by four quick closes. (In this case, the program being traced was /bin/sh, so it is probable that the closes were associated with a fork. This could be verified by further tracing if desired.)

Trace freezing. Another frequent requirement is to enable or disable the kernel tracing process on a global basis. The freeze command will halt all trace collection until an unfreeze command is issued. This feature can be used to limit the collected data to the duration of an experiment.

```
(kitrace) trace sync+50 REG g1
(kitrace) clear
(kitrace) print
_sync    10:33:36.342703 (        ) pid 7759
_sync+50 10:33:36.342803 (0.000100) g1=f80a2424 (_spec_sync)
_sync+50 10:33:36.400674 (0.057871) g1=f80beb90 (_ufs_sync)
_sync+50 10:33:38.117703 (1.717029) g1=f802ad28 (_nfs_sync)
_sync+50 10:33:38.135095 (0.017392) g1=f80922e0 (_rf_sync)
_sync+50 10:33:38.135175 (0.000080) g1=f80a8b04 (_tmp_sync)
_sync+50 10:33:38.135221 (0.000046) g1=f8008e18 (_hsfs_sync)
_sync+50 10:33:38.135266 (0.000045) g1=f80e0540 (_null_sync)
_sync+50 10:33:38.135375 (0.000109) g1=f80f47f0 (_flfs_sync)
_sync+50 10:33:38.135422 (0.000047) g1=f810f484 (_fpfs_sync)
...
_sync+50 10:33:38.135563 (0.000047) g1=f811a51c (_umap_sync)
_sync+50 10:33:38.135615 (0.000052) g1=f80e29fc (_select_sync)
```

Figure 3. Capturing registers in symbolic form

```
(kitrace) delete sync sync+50
(kitrace) clear
(kitrace) trace open ONLYPID 7372 close ONLYPID 7372
(kitrace) print
_open          Nov 10 10:33:45.673123 (          )
_close         Nov 10 10:33:45.673836 (   00.000713)
_close         Nov 10 10:33:45.673999 (   00.000163)
_close         Nov 10 10:33:45.674163 (   00.000164)
_close         Nov 10 10:33:45.674336 (   00.000173)
```

Figure 4. Limiting tracing to a specific process ID

Freezing can also be enabled and disabled dynamically under the control of selected tracepoints. For example, the `sync` command traced above may require disk or network access. If we want to trace the behavior of disk operations, we cannot use the `ONLYPID` option because disk interrupts are not associated with a particular process. Instead, we can use the `FREEZE` and `UNFREEZE` to enable tracing only for the duration of the `sync`, increasing the probability of catching only relevant data. An illustration of this appears in Figure 5. We can see that the `sync` operation rapidly makes a large number of calls to the disk strategy routine `sdstrategy`. In fact, there are so many calls that internal queuing limits are reached and the first I/O operation completes (sdintr) before all I/O has been scheduled. We do not see the completion of all of these operations, because the `sync` call itself completes (`sync+70`) before all scheduled I/O takes place.

Incidentally, this trace also allows us to evaluate the effectiveness of the disk head-scheduling algorithm by recording the elapsed times of a sequence of write operations. If the scheduling is effective, the times will be consistent and close to the minimum cited in the manufacturer's specifications. If it is ineffective, inconsistent times across the possible range of seek times should be observed.

There is also a `LATEFREEZE` option, which delays freezing by a certain number of traces in a manner similar to the delayed triggering of hardware logic analyzers. This could be used, for example, to attempt to capture the completion times of more of the disk I/O's caused by `sync` operation, as in Figure 6. This shows a few of the many rapid disk I/O's (presumably writes) that complete within a short time after the `sync` operation returns to the caller.

Time base selection. In all the above examples, `kitrace` gave time differences between adjacent traces. Sometimes it is more useful to see differences from some base time. For example, the frequent calls to `_sdstrategy` in the above traces make it difficult to see the total amount of time spent doing disk I/O. It might be more useful to know the total time elapsed between the beginning of the `sync` operation and the completions of the various disk I/O's. While this could be done by hand, by subtracting timestamps, `kitrace` provides a time-base facility to ease the task. The `BASE` option informs `kitrace` that a tracepoint should serve as a base for calculation of future time intervals. Figure 7 uses this feature to show that the `sync` operation itself took over two seconds just to finish scheduling disk I/O, and that rapid disk operations continued for another 65 ms after that, without risking an overfull trace buffer due to unrelated disk activity. Although some of them are not shown in the figure,

```
(kitrace) freeze
(kitrace) clear
(kitrace) delete open close
(kitrace) trace sync UNFREEZE
(kitrace) trace sync+50 REG g1
(kitrace) trace sync+70 FREEZE
(kitrace) trace sdstrategy PID sdintr
(kitrace) print
_sync       10:33:59.771442 (       ) UNFREEZE
_sync+50    10:33:59.771539 (.000097) g1=f80a2424 (_spec_sync)
_sdstrategy 10:33:59.819478 (.047939) pid 7840
_sdstrategy 10:33:59.820051 (.000573) pid 7840
_sdstrategy 10:33:59.820297 (.000246) pid 7840
_sdstrategy 10:33:59.820627 (.000330) pid 7840
...
_sync+50    10:33:59.842798 (.000287) g1=f80beb90 (_ufs_sync)
_sdstrategy 10:33:59.843206 (.000408) pid 7840
_sdintr     10:33:59.862006 (.018800)
_sdintr     10:33:59.869592 (.007586)
_sdintr     10:33:59.887792 (.018200)
_sdintr     10:33:59.897314 (.009522)
...
_sdstrategy 10:34:01.697801 (.000476) pid 7840
_sdintr     10:34:01.705756 (.007955)
...
_sync+70    10:34:01.870146 (.000053) FREEZE
```

Figure 5. Automatically freezing and unfreezing tracing

there were only 23 I/O completions recorded in this interval in the example, so that in fact freezing never took effect. Because of the long delay (over 5 seconds) between the last write completion (`_sdintr`) and the next call to `_sync`, we can be sure that we have observed all I/O resulting from the `sync`.

`Kitrace` also provides an `ADJACENT` option, which can be applied to a tracepoint to cancel the constant time base and return to printing time differences between adjacent traces.

Other tracing options. `Kitrace` offers several other options that can be applied to tracepoints:

1. `PSR` collects the processor status register, in a manner analogous to the `REG` option.
2. `ALLREGS` collects all processor registers. This can be useful in some circumstances, but slows trace collection noticeably.
3. `CONST` *value* stores a constant value that was specified in the `trace` command. This can be useful to distinguish traces generated by different `trace` commands targeted at a single instruction.

```
(kitrace) delete sync+50
(kitrace) trace sync UNFREEZE PID sync+70 LATEFREEZE 50
kitrace:  Tracepoint at _sync already exists, updating
kitrace:  Tracepoint at _sync+0x70 already exists, updating
(kitrace) freeze
(kitrace) clear
(kitrace) print
_sync       10:34:11.641381 (        ) UNFREEZE pid 7875
_sdstrategy 10:34:11.689315 (0.047934) pid 7875
_sdstrategy 10:34:11.689890 (0.000575) pid 7875
...
_sdintr     10:34:11.709694 (0.001417)
_sdintr     10:34:11.717275 (0.007581)
...
_sync+70    10:34:13.669994 (0.016060) LATEFREEZE
_sdintr     10:34:13.677172 (0.007178)
_sdintr     10:34:13.686267 (0.009095)
...
_sdintr     10:34:14.182041 (0.032128)
_sdintr     10:34:14.221724 (0.039683)
...
```

Figure 6. Freezing trace capture after a delay

4. `STACK` *offset* collects a word from the processor stack at a given offset. This can be used to determine return addresses or arguments.
5. `FRAME` *offset* collects a word from the specified offset within the current stack frame. This can be used to examine local variables.
6. `LOC` *address* collects a word from the specified address. This can be used to record the value of a global variable at a given time.

Other commands

Most of the commands provided in interactive mode (`trace`, `print`, `delete`, `clear`, `freeze`, and `unfreeze`) have already been discussed. In addition, `kitrace` provides the following commands:

1. `list` lists the current tracepoints, one per line, including all options that were specified when they were set.
2. `printclear` is equivalent to `print` followed by `clear`, except that it guarantees that no traces will be lost between the two operations.
3. `follow` is equivalent to typing `printclear` whenever there is at least one trace in the buffer. It is useful for avoiding buffer overruns in situations where there is a lot of data to be collected.
4. `quit` exits from `kitrace`.

```
(kitrace) clear
(kitrace) trace sync BASE UNFREEZE
kitrace:  Tracepoint at _sync already exists, updating
(kitrace) print
_sync       10:34:55.254005 (        ) UNFREEZE BASE
_sdstrategy 10:34:55.301865 (0.047860) pid 7915
_sdstrategy 10:34:55.302434 (0.048429) pid 7915
...
_sdintr     10:34:55.343381 (0.089376)
_sdintr     10:34:55.350939 (0.096934)
...
_sdstrategy 10:34:57.495028 (2.241023) pid 7915
_sync+70    10:34:57.511464 (2.257459) LATEFREEZE
_sdintr     10:34:57.528244 (2.274239)
_sdintr     10:34:57.556774 (2.302769)
...
_sdintr     10:34:58.154117 (2.900112)
_sync       10:35:03.663681 (8.409676) UNFREEZE BASE
...
```

Figure 7. Modifying the time base

Implementation

`Kitrace` is implemented with a small add-on kernel package and a larger user-level interface program. The kernel code is designed to be as minimal as possible, with the bulk of the complexity kept in the user-level program for ease of maintenance.

Kernel code

The kernel support is divided into a small amount of machine-dependent assembly code and a larger machine-independent C routine. The assembly code has been implemented (at various times) on a Motorola 68010,[19] an Intel 80386,[20] and a SPARC.[21] The implementation described here is for the SPARC, since that is the most recent and most complex version of `kitrace`. The SPARC version also supports dynamic loading of the kernel code, so that `kitrace` can be used on an unmodified kernel.

A tracepoint is created (by the user-level program) by filling in an entry in a control table and replacing the traced opcode with a trap or breakpoint instruction. The control table encodes information about what data should be recorded when the tracepoint is reached, data needed to continue execution after the trace is collected, and an active flag that is used to avoid race conditions in the algorithms described below. When a traced instruction is executed, the trap causes the processor to jump to the assembly code, which stores all registers, sets up a C environment, and calls the machine-independent part of the kernel support. The machine-independent code then searches the control table (based on the address of the traced instruction) and collects data based on the requests in the control entry.

One critically important datum that is always collected is the current time in microseconds. The precise timing provided by `kitrace` requires a high-resolution timer, supporting a granularity of 10 μs or better, that can be read with low overhead. Many kernels provide a subroutine that will provide this information. On other machines, it has been necessary to write a special routine for this purpose. Usually, since the periodic clock interrupt is generated by a simple clock/counter chip, it is possible to read the current count from the chip and convert this into a sufficiently accurate time measurement.

Once the data has been collected, the C routine returns to the assembly code. Since the opcode that was replaced by the trap still needs to be executed, it is re-inserted in its original location and executed in a single-step mode appropriate to the processor. After single-stepping, the assembly code regains control and can then reinsert the trap so that tracing will occur the next time the traced instruction is executed. Finally, normal kernel execution is resumed.

An alternative to single-stepping the traced instruction would be to branch to a specially-constructed instruction sequence in a temporary buffer. This sequence could emulate the effects of the traced instruction, usually by duplicating it, and then jump back to the instruction following it in sequence. On most processors, this would be more efficient than single-stepping, which necessarily introduces an extra trap sequence. We have not implemented this option due to its increased complexity, but it is an interesting direction for future research.

User-level code

The user-level part of `kitrace` consists of a single C program that interfaces to the kernel via `/dev/kmem` and the symbol table from the bootable kernel, `/vmunix`. The kernel code maintains certain global variables that make it possible to locate and manipulate the trace tables. These include pointers to the trace control and trace data buffers, a monotonically-increasing counter of the number of traces, and verification information that allows the user-level code to be sure it is compatible with the kernel version.

Tracepoint control. Tracing is controlled by the previously-mentioned array of structures that describe tracepoints. To set a tracepoint, `kitrace` first fills in a new entry in this table and writes it to kernel space. It then rewrites the kernel variable that records the number of entries in the table, making the new entry visible to the kernel code. Note that there is no possibility of race conditions here, since the table entry exists before it becomes visible to the kernel, and the table entry will not be needed because there is no trap instruction yet. Once the control table has been updated, `kitrace` replaces the traced opcode with a trap instruction. This enables tracing at the given location.

If it is necessary to update a current tracepoint, a five-step process is followed to prevent races:

1. Replace the breakpoint instruction at the trace location with the original opcode.
2. Clear the active bit in the flags word (this is a single memory operation, so it cannot cause a race).
3. Rewrite the entire control entry.
4. Reset the active bit.
5. Replace the breakpoint instruction.

This operation can cause traces to be lost; we have found this to be acceptable in our usage. If lost traces were a problem, it would be a simple matter to prevent them through a more complex procedure involving creating a duplicate control entry before the original is modified.

Trace deletion proceeds in a similar race-free fashion. First the trap instruction is replaced with the original opcode, making the control-table entry superfluous. Control-table entries following the deleted one are then packed downwards. To avoid races, each entry is copied from its old position to the new one using a three-step process:

1. Clear the active bit of the entry being replaced. This will cause the kernel to ignore this entry when searching the control table. No traces will be lost because there is still an active copy of the replaced entry in some other (earlier) slot in the table.
2. Write the new entry to the deactivated table slot, being sure that the new entry also has a clear active bit.
3. Reset the active bit in the replacement entry.

Note that, after this process has executed for a particular control-table entry, there will be two copies of that entry in the table. This does not cause problems because the entries are identical and the kernel will simply obey the entry it finds first.

Trace buffer manipulation. Manipulation of the trace buffer proceeds in a similar fashion. The kernel fills the buffer in a circular manner, overwriting old entries if necessary. Each entry contains the address traced, the time the trace was collected, a copy of the control-table information indicating what data was collected, and the captured data itself. A monotonically-increasing counter, `kit_count`, indicates the number of traces captured since the last boot. Taking this counter modulo the buffer size gives a pointer to the oldest trace record, which is also the next to be collected. The user-level interface program also maintains a count of traces seen, `kit_last_count`. This counter is stored in the kernel so that it will persist across invocations of the interface program, but it is never modified by the kernel.

The `print` command reads `kit_count` and the trace buffer. It then rereads the trace counter and compares it to the first value read. If the two values are equal, no traces have been collected while the buffer was being read, so it is known to be in a consistent state. If `kit_count` exceeds `kit_last_count`, there are printable traces in the buffer. If the difference between the two counters is greater than the buffer size, a warning is printed indicating that traces are lost. The trace buffer contains enough information to correctly print results even if the tracepoint has since been deleted, so printing is simply a matter of formatting the appropriate entries.

If `kit_count` changes between the two times it is read, then new traces were added to the buffer while it was being read. There is no way for the user-level program to know whether it captured the old or the new values of these buffer entries, so it assumes that they are in an unknown state and refuses to print them. A future print operation will be certain to capture the new values and print them correctly. An important detail occurs when the trace buffer has overflowed (indicated by `kit_count` exceeding `kit_last_count` by more than the buffer size. In this case, even though all of the traces in the buffer are valid, the ones representing the difference between the two values of `kit_count` cannot be printed because there is no way to determine whether a particular entry is old or new (or even a mix). However, the buffer-clearing algorithm will not remove unprinted new entries, so this is not a serious drawback.

To clear the trace buffer, the user-level code simply writes its copy of the last-trace-entry pointer, `kit_count`, into the last-entry-seen pointer, `kit_last_count`. If clearing is combined with printing, this ensures that only those traces that have been printed will be marked as having been seen, because `kit_count` was collected before the trace buffer was read.

```
(kitrace) delete sdstrategy sdintr sync+70
(kitrace) list
_sync          UNFREEZE BASE
(kitrace) clear
(kitrace) unfreeze
(kitrace) trace sync sync+4 sync+8 sync+0xc sync+10
kitrace:  Tracepoint at _sync already exists, updating
(kitrace) print
_sync          Nov 10 10:35:45.165061 (          )
_sync+4        Nov 10 10:35:45.165138 (   00.000077)
_sync+8        Nov 10 10:35:45.165183 (   00.000045)
_sync+c        Nov 10 10:35:45.165240 (   00.000057)
_sync+10       Nov 10 10:35:45.165285 (   00.000045)
_sync          Nov 10 10:35:47.262426 (   02.097141)
_sync+4        Nov 10 10:35:47.262502 (   00.000076)
_sync+8        Nov 10 10:35:47.262547 (   00.000045)
_sync+c        Nov 10 10:35:47.262604 (   00.000057)
_sync+10       Nov 10 10:35:47.262649 (   00.000045)
```

Figure 8. Measurement of the cost of tracepoints

The `follow` command is implemented as a loop, alternating buffer-print and buffer-clear operations. Although this monopolizes the CPU (since there is no way to pause until the kernel has made new trace entries), we have found that appropriate priority adjustment with the `nice` command still makes `follow` a useful tool.

Performance

It is very easy to use `kitrace` to measure its own performance. Figure 8 shows how tracepoints on successive instructions can be used to measure the overhead introduced by `kitrace`. In this measurement. performed on a Sun IPC workstation, a single tracepoint requires from 45 to 77 μs, including the execution time of the instruction traced. The variability is due partly to clock inaccuracies, and partly to the way `kitrace` handles register windows on the SPARC, discussed below. Note that our performance-measurement method is not itself dependent on kernel internals. Any short sequence of instructions can be used to measure the performance of `kitrace`, so long as there are no branches into or out of the middle of the sequence.

Figure 8 shows that the impact of adding a tracepoint to an execution path is minimal. Nevertheless, it is best to factor the cost of tracing into any analysis, as large numbers of tracepoints can introduce a noticeable delay of their own. Fortunately, the repeatability of `kitrace`'s measurement overhead makes it easy to calculate a correction based on the total number of traces collected.

Figure 9 gives histograms showing the distributions of timings measured for the first four instructions of the `getpid` system call on the same Sun IPC workstation. In each figure,

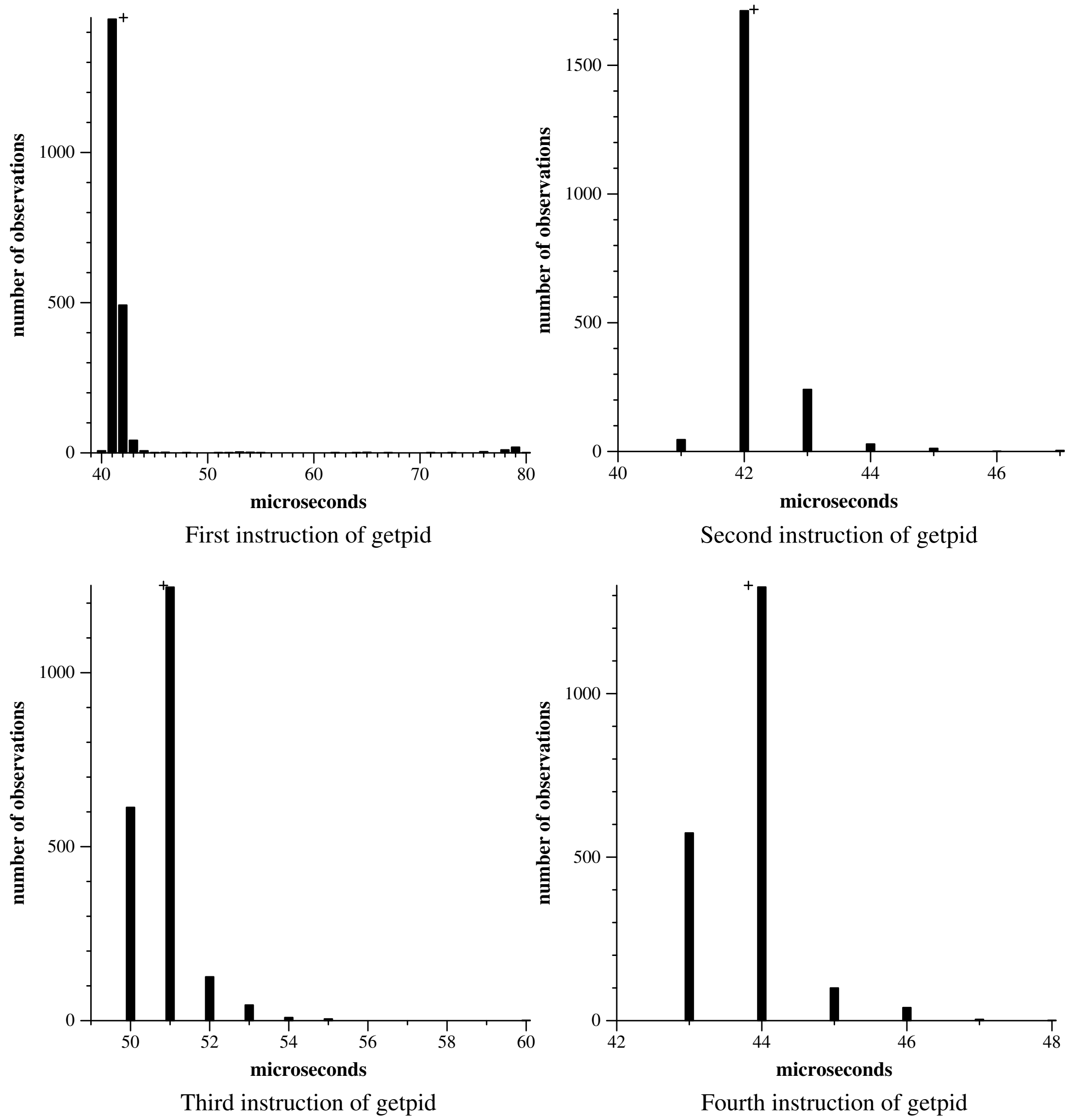

Figure 9. Timing observations for first four instructions of `getpid`

Table III. Means, 99 per cent confidence intervals, and standard deviations for the costs of four tracepoints

Instruction	Mean (μs)	Standard Deviation (μs)
First	42.06±0.29	5.04
Second	42.15±0.03	0.53
Third	50.83±0.04	0.73
Fourth	43.82±0.04	0.63

the mean of the samples is indicated by a small cross near the top of the graph. Table III gives estimates of the standard deviations and 99% confidence intervals for each of these instructions.

There are several reasons for the timing variations shown in the graphs in Figure 9:

1. Variations in the execution times of the instructions themselves (these are negligible on the SPARC).
2. Operating-system support overhead, such as the cost of spilling SPARC register windows.
3. Inaccuracies in clock readings.[22]
4. Interrupts that occur between two instructions (these are made more likely by the fact that interrupts are turned off during trace collection, effectively increasing the execution time of the instructions and thus the probability of being interrupted).

Of these, item 2 is the cause of most variations, in particular the occasional large variation seen in the first instruction of `getpid`, seen in the histogram and also in the large standard deviation of the sample time for this instruction.

SPARC register spilling

On many architectures, it is a simple matter to set up a C environment as part of trace collection. The usual process involves saving a few registers on the kernel stack, which is always in memory, and loading a new frame pointer. The SPARC processor, however, requires a much more complicated process, detailed in Appendix D of reference 20. The SPARC features a 'register window' architecture, where a small portion of a large register file provides the accessible machine registers, called the *register window*. A *current window pointer* identifies the portion of the register file currently in use. When a trap or subroutine call occurs, the current window pointer is decremented to point to a new part of the register file. This saves the overhead of preserving registers on the stack. At a later time, the original registers can be restored by incrementing the current window pointer.

When there is no more room in the register file, a *window overflow* trap occurs and a *register spilling* algorithm is invoked. At this time, one or more sets of machine registers are copied to the stack to make room for new register windows. This can be a complex process, since there is no guarantee that the virtual memory reserved for this purpose is mapped or even resident. At a later date, when the current window pointer is incremented to point to an empty window, a *window underflow* trap occurs and causes the appropriate registers to be reloaded.

This architecture introduces a large variation into the overhead of `kitrace` tracepoints, because the contents of the register file are dependent on the execution history of the processor.

For example, if there have recently been several nested subroutine calls without corresponding returns, the register file will be full and a window overflow will occur when the `kitrace` trap is reached and on every subroutine call involved in tracing. This will cause at least a few microseconds to be lost while registers are copied to the stack, and can potentially cause a much larger delay if the appropriate stack page needs to be mapped or even paged in.

Conversely, if a deeply-nested set of subroutines has just terminated, the register file will be almost completely empty, and the tracing code will be able to execute faster because it will not have to spill any registers at all.

These effects can be seen in the large standard deviation shown in Table III for the first instruction of `getpid`. The second through fourth instructions have much smaller standard deviations because the register spilling does not need to be repeated for tracepoints which occur so close together.

Multi-machine tracing

One advantage of `kitrace` is that it collects very precise timestamps. This naturally leads to the idea of using `kitrace` to analyze operations on multiple machines. For example, one could watch a client machine issue an NFS operation, follow the progress of that operation on a server, and then observe the resulting behavior of the client.

Unfortunately, this is more easily said than done. It is quite straightforward to take two traces and combine them (using `sort`) into a single output file; if necessary, the output lines can also be tagged with the identity of the machine that originally generated them. It is only slightly more difficult to write an `awk` or `perl` script that recalculates the time-difference field. However, the two traces will only be combined correctly if the clocks on the two machines are very precisely synchronized.

In most modern networks, clocks are kept synchronized using the Network Time Protocol[23] or a similar method. However, these protocols do not generally synchronize beyond an accuracy of a few milliseconds, and machines must be up for a significant time (as much as several hours) to achieve good synchronization. We have found that this level of performance is inadequate for synchronizing most client/server traces.

The ideal solution to this problem, of course, would be to achieve better automated time synchronization. `Kitrace` often requires synchronization to the level of a single Ethernet delay, which (for short packets and fast processors) can be as little as 100 μs. Some recent researchers[24, 22] have reported improvements on NTP which can achieve accuracies in these ranges, but their work was not available to us when `kitrace` was being developed.

Instead, we have chosen an *ad hoc* solution based on our knowledge of the problem. The idea is to have the client site perform one or more operations that will be clearly identifiable on the server. In general, almost anything will do that is not part of the main test and that does not regularly happen in background, such as the `link` operation. By matching these operations and their listed times, it is possible to calculate an approximate clock differential between the two machines. A simple `awk` script can then adjust the clock in one of the trace files before they are combined.

For many purposes, however, even this process is not sufficient. The method assumes exact simultaneity for two operations which are actually separated in time, and relies on the premise that the error introduced by this assumption is less than the synchronization accuracy required. Often, the processing involved on both sides of the network connection is variable enough to invalidate this premise. One solution would be to add a special-purpose synchronization operation that performed a remote call to the server with as little overhead as possible. However,

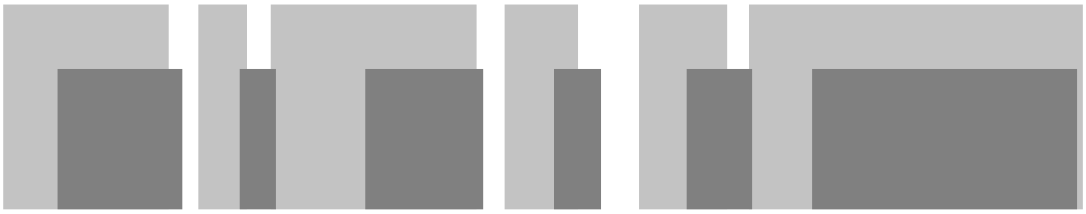

Figure 10. Display of misaligned operations

we do not wish to modify the kernel any more than absolutely necessary.

Instead, we have found that it is possible to synchronize clocks empirically. After using the above method to achieve approximate synchronization, the collected data files from client and server are fed into a TCL[25, 26] program that displays each operation as a shaded rectangle (see Figure 10). When the clock offset is incorrect, the rectangles will overlap. A correct offset will produce nesting rectangles (see Figure 11), where the time taken by a server-side operation is a proper subset of the time needed on the client. By dragging with the mouse, the rectangles can be made to nest, and the TCL program will then print the associated offset. We have found that this method is easy to use and allows clocks to be aligned to within a few hundred microseconds.

Debugging

`Kitrace` was designed as a performance-measurement tool, but it has also proved useful in kernel debugging. Although basic breakpoint-and-step debuggers continue to be the mainstay of daily bug-finding, the ability to dynamically collect information without significantly disturbing the kernel has complemented these debuggers in a very handy way. The following `kitrace` features, unavailable with traditional debugging tools, can be useful in finding kernel problems:

1. The ability to examine kernel behavior without significantly affecting timing.
2. The ability to collect data about infrequent events over a long period, for later analysis.
3. The ability to enable tracing when some unusual event occurs. (The `LATEFREEZE` option is often useful for this.)
4. The ability to examine the relative timing of events involved in race conditions.

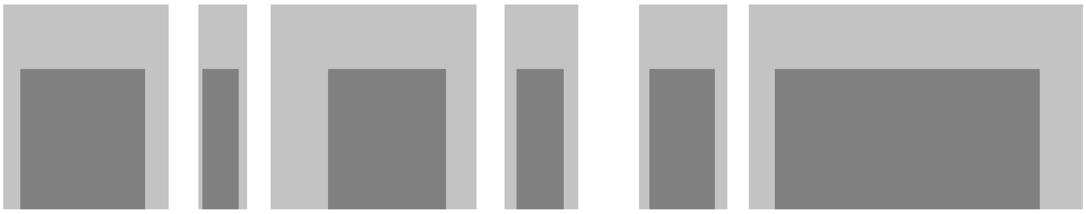

Figure 11. Display of time-aligned operations

Kitrace is also useful for quick non-intrusive peeks at the behavior of kernel code. A tracepoint can be set, examined, and deleted without the overhead of invoking a full-scale debugger. This is often preferable to halting the kernel, setting a breakpoint, and thus disrupting all other work in progress on the machine.

LESSONS

Kitrace has proven to be a far more useful tool than originally expected. As it has been used and ported to various machines, the implementation has been improved and simplified. Nevertheless, there are still a few flaws:

- The current version is unabashedly assembly-oriented. Although it has knowledge of kernel symbols, the user is forced to spend a good deal of time disassembling code to determine the proper location of tracepoints,* which registers to collect, and the offsets of critical variables in the stack frame.
- It is easy to accidentally crash the kernel by misusing the tool (e.g., setting a tracepoint in data space).
- Since we have chosen to place the burden of avoiding race conditions entirely on the user-level program to improve portability and performance, errors in this code can have an adverse effect on the reliability of results reported by kitrace.
- The follow command is implemented using a CPU polling loop, which causes the program to interfere significantly with the behavior of the kernel being traced when this command is used.
- The user-level/kernel interface is dependent on the quality and reliability of the /dev/kmem driver.
- Because the user- and kernel-level code rewrites instruction opcodes, kernel text-space write protection must be disabled in kernels that support kitrace capabilities. This introduces a slight risk of failing to catch certain types of pointer errors.
- Some types of multi-machine tracing require more accurate clock synchronization than can be achieved without special-purpose hardware.

The lack of source-level access could be cured by folding in appropriate code taken from a debugger such as gdb.[27] The risk of crashing the kernel is fundamentally insoluble, although adding a source-level interface would probably help the problem. The clock-synchronization problem cannot be solved generally without further research. The remaining difficulties could be alleviated by rewriting the kernel support as a device driver. However, this would increase the entanglement of the code with the kernel, with a related impact on the ability of the program to trace those kernel routines (right now, kitrace tries very hard to use as little of the kernel as possible, so that tracepoints can be set in a wide range of locations). We have not yet decided whether to go forward with the driver idea.

RELATED WORK

Kernel tracing facilities

Kitrace is most closely related to the idea of building trace calls into an operating-systems kernel at compile time. However, kitrace is so much more convenient that it is

* The info line command of gdb is very helpful with this.

typically used in a very different way. Built-in tracing implementations are limited to collecting information at preselected points. The only interactive control lies in the ability to dynamically enable selected tracepoints. An expensive kernel recompilation and reboot is needed to add a previously-unanticipated trace. In addition, built-in tracing implementations tend to be more heavyweight than `kitrace`.

On the other hand, the source-level coding of built-in kernel tracing offers data-collection power that `kitrace` cannot match. Also, kernels can be shipped to naive end users with a few carefully-selected built-in tracepoints ready to be enabled if performance problems arise.

These differences make `kitrace` appropriate for low-level kernel development, while built-in tracing is more suitable to production environments. For the developer who is intimately familiar with the kernel, the ease of setting up and and modifying experiments far outweighs the more limited functionality, while the end user will place far more value on the convenience of pre-specified tracepoints.

Profiling

As mentioned earlier, profiling has long been a popular tool for kernel performance measurement. However, we believe that profiling is a very limited technique, and have found that `kitrace` is useful in a much wider variety of circumstances.

Profiling has two major drawbacks for kernel tuning: it is a statistical technique,[2] and it is inherently limited to examining only CPU performance. While the former drawback can be minimized by methods such as the use of large samples, the latter is inescapable. Profiling is invaluable for locating 'hot spots,' and can sometimes be used to identify lock contention, but it is of no help whatsoever in examining delays caused by paging, queuing, excessive I/O, or scheduling conflicts. Even when dealing with CPU hot spots, we have found that it is often best to locate the offending routines with a profiler and then use `kitrace` to analyze the internal behavior of the algorithm. The interactive flexibility of `kitrace` makes tracing of small sections of code simpler and quicker than the alternative of breaking a routine into small pieces and recompiling for another profiling run.

Hardware analyzers

The features and capabilities of `kitrace` are very close to those available from some hardware-based analyzers.[5, 6] Hardware analysis generally offers even more power and can display more detailed information than `kitrace`. Balanced against these strengths are the lower convenience and higher cost of a hardware-based approach.

FUTURE WORK

As mentioned above, one of the biggest flaws in `kitrace` is the lack of source-level tracing support. We have considered integrating code from `gdb` to allow specification of tracepoints on a source-line basis, and to allow collection of arbitrary source-program data. However, this is a nontrivial change. The symbolic-debugging routines are very complex, so that it would not be easy to integrate them. Also, the current kernel support is very limited in its ability to collect data (it cannot even follow a pointer, let alone the sort of chain of pointer-offset-index tuples that would be needed for a general data-collection facility). For these reasons, we have decided not to pursue a source-level interface at present.

Other possible improvements would be the device-driver conversion and the improved single-stepping method alluded to above, a graphical trace-display tool similar to Traceview[28, 29] and a generalized time-synchronization facility, either automatic or graphical (the current TCL-based program is specialized to a particular client-server problem that we needed to investigate).

One final idea of interest would be the extension of `kitrace` capabilities to user-level processes. Although the vagaries of schedulers make the precise time-measurement facilities of `kitrace` less interesting than for the kernel, many of the other advantages of the tool would be equally applicable to user-level debugging. We believe that it would not be difficult to provide such a capability, although it would require more extensive kernel modifications than those needed by the current tool.

HISTORY

`Kitrace` was originally written for a Sun-2 workstation in the summer of 1985, to analyze performance problems in a mini-supercomputer kernel. That version was developed in a couple of days, taking advantage of the friendly architecture of the 68000 and the availability of kernel sources. Most of the important features of the current version were added over the following month.

The original version was lost with the collapse of the company for which it had been developed. A complete rewrite was done a few years later, when the need for the facility became critical. This version was then ported to the SPARC* and the remaining features described here were added.

The SunOS 4.1.1 version for the SPARC was implemented without reference to kernel source code, so that it will be free of licensing problems. This version is publicly available via anonymous `ftp` from `ftp.cs.ucla.edu`, in the directory `pub/ficus/geoff/kitrace`.

ACKNOWLEDGEMENTS

This work was partially supported by the Advanced Research Projects Agency under contract N00174-91-C-0107. Work on earlier versions of `kitrace` was performed at Culler Scientific, Inc., and at Locus Computing Corporation.

I am indebted to John Heidemann for his comments on earlier versions of this paper, his willingness to experiment with `kitrace` and suggest for improvements and new functionality, and his frequent encouragement. I am also thankful to the anonymous reviewers for their helpful suggestions, which greatly improved the clarity of the presentation.

REFERENCES

1. Susan L. Graham, Peter B. Kessler, and Marshall Kirk McKusick, ‘An execution profiler for modular programs’, *Software—Practice and Experience*, **13**(8), 671–685 (1983).
2. Dominic A. Varley, ‘Practical experience of the limitations of gprof’, *Software—Practice and Experience*, **23**(4), 416–463 (1993).
3. J. Bradley Chen, ‘Software methods for system address tracing’, *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa, California, October 1993, pp. 178–185. IEEE.
4. James R. Larus, ‘Efficient program tracing’, *IEEE Computer*, **26**(5), 52–61 (1993).
5. Uwe Kleinhans, Joerg Kaiser, and Karol Czaja, ‘Spearmints: Hardware support for performance measurement in distributed systems’, *IEEE Micro*, **13**(5), 69–78 (1993).

* The SPARC, not incidentally, was by far the most difficult port yet done, requiring two full weeks just to get the assembly code to run.

6. Charles Melear, 'M88000 software tools', *WESCON/89 Conference Record*, Ventura, CA, November 1989, pp. 738–742. IEEE, Electronic Conventions Management.
7. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Sherriff, and John K. Ousterhout, 'Measurements of a distributed file system', *Proceedings of the Thirteenth Symposium on Operating Systems Principles*. ACM, October 1991, pp. 198–211.
8. Gerald P. Bozman, Hossein H. Ghannad, and E. D. Weinberger, 'A trace-driven study of CMS file references', *IBM Journal of Research and Development*, **35**(5–6), 815–828 (1991).
9. Helmar Burkhart and Roland Millen, 'Performance-measurement tools in a multiprocessor environment', *IEEE Transactions on Computers*, **38**(5), 725–731 (1989).
10. Joseph P. CaraDonna, 'Measuring lock performance in multiprocessor operating system kernels', *Proceedings of the Symposium on Experience with Distributed and Multiprocessor Systems*, San Diego, CA, January 1993, pp. 37–56. USENIX.
11. Paul S. Dodd and Chinya V. Ravishankar, 'Monitoring and debugging distributed real-time programs', *Software—Practice and Experience*, **22**(10), 863–867 (1992).
12. James Griffioen and Randy Appleton, 'Reducing file system latency using a predictive approach', *Proceedings of the Summer USENIX Conference Proceedings*, Boston, MA, June 1994. USENIX.
13. John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, 'A trace-driven analysis of the UNIX 4.2 BSD file system', *Proceedings of the Tenth Symposium on Operating Systems Principles*. ACM, December 1985, pp. 15–24.
14. Dave Plauger, 'Timing is everything: Real-time UNIX', *Mini-Micro Systems*, **22**(2), 72–76 (1989).
15. Peter L. Reiher, Steven Bellenot, and David Jefferson, 'Debugging the Time Warp Operating System and its application programs', *Proceedings of the Symposium on Experience with Distributed and Multiprocessor Systems II*, Atlanta, Georgia, March 1991, pp. 203–220. USENIX.
16. Richard G. Guy, *Ficus: A Very Large Scale Reliable Distributed File System*, Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.
17. Richard G. Guy, Gerald J. Popek, and Thomas W. Page, Jr., 'Consistency algorithms for optimistic replication', *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.
18. Sun Microsystems, 'NFS: network file system protocol specification', Technical Report RFC-1094, Internet Request For Comments(March 1989).
19. Prentice Hall, Englewood Cliffs, NJ, *M68000 8-/16-/32-Bit Microprocessors User's Manual*, seventh edition, 1989.
20. Intel Corporation, Santa Clara, CA, *80386 Programmer's Reference Manual*, 1986.
21. Sun Microsystems, Inc., Mountain View, CA, *The SPARC^{TM} Architecture Manual*, January 1991. Version 8.
22. David L. Mills, 'Precise synchronization of computer network clocks', *ACM Computer Communication Review*, **24**(4), 28–43 (1994).
23. Dave L. Mills. Internet time synchronization: The network time protocol. Network Working Group Request for Comments: 1129, October 1989.
24. K. Arvind, 'Probabilistic clock synchronization in distributed systems', *IEEE Transactions on Parallel and Distributed Systems*, **5**(5), 474–487 (1994).
25. John K. Ousterhout, 'TCL: An embeddable command language', *USENIX Conference Proceedings*. USENIX, January 1990, pp. 133–146.
26. John K. Ousterhout, 'An X11 toolkit based on the TCL language', *USENIX Conference Proceedings*. USENIX, January 1991, pp. 105–115.
27. Free Software Foundation, Cambridge, MA, *GDB 4.9 Reference Manual*, April 1993.
28. Allen D. Malony, David H. Hammerslag, and David J. Jablonowski, 'TraceView: A trace visualization tool', *Proceedings of the First International ACPC Conference*, ed., Hans P. Zima, Salzburg, Austria, September 1991, pp. 102–114. Springer-Verlag.
29. Allen D. Malony, David H. Hammerslag, and David J. Jablonowski, 'TraceView: A trace visualization tool', *IEEE Software*, **8**(5), 19–28 (1991).