FAST KERNEL TRACING: A PERFORMANCE EVALUATION TOOL FOR LINUX

ROBERT D. RUSSELL
Department of Computer Science
University of New Hampshire
Durham, New Hampshire 03824-3591, USA
Email: rdr@unh.edu

MRINALINI CHAVAN
EMC Corporation
80 South Street
Hopkinton, Massachusetts, USA
Email: chavan_mrinalini@emc.com

ABSTRACT

This paper describes a new software performance evaluation tool for Linux called FKT: Fast Kernel Tracing. This tool consists of a number of modifications and additions to the Linux kernel on Pentium PCs that provide a mechanism for recording the flow of control through the kernel with very high precision and very low perturbation to kernel timings. This is accomplished by utilizing special instructions in the Pentium hardware to record the value of the processor's cycle clock and to permit rapid access to shared memory buffers. Data is collected in real-time during a measurement session, and is then analyzed off-line to produce detailed accountings of processor activities.

The primary design goal for this tool is to provide a simple, highly-accurate, low-overhead technique for evaluating a kernel's performance. Its use to investigate the TCP-IP protocol stack in Linux will be discussed. It has also proven useful for two other unrelated tasks: kernel debugging and educational instruction, as will also be discussed.

Keywords: performance evaluation, kernel probing, operating systems.

1 INTRODUCTION

FKT is a software tool for accurately evaluating the performance of Linux kernels running on Pentium PCs. It consists of a number of modifications and additions to the Linux kernel, as well as some user-level programs to control the probing and to perform off-line analysis.

The basic motivation for FKT is the observation that it is extremely difficult to know exactly what an operating system is doing, especially on a multiprocessor platform or in a networked environment [1]. Furthermore, it is extremely difficult to characterize this activity in a way that is useful to a kernel programmer.

FKT is a method for obtaining a fine-grained trace of the dynamic flow through an operating system on a uni- or multi-processor platform. The method consists of placing special software "probe" macros at various locations in the kernel source code. Placement of probes is controlled by the kernel programmer, as is any extra information recorded by the probe. Typically one probe is placed at the entry to, and another probe at the exit from, every kernel function of interest to the programmer. By default, probes are always placed at the entry to and exit from every system call, trap, interrupt and process switch in the kernel. Additional probes will be placed at the entry to and exit from each major routine in the sections of the kernel that are to be evaluated.

For example, to evaluate the TCP-IP protocol stack, probes are placed in the socket interface, the TCP layer, the IP layer and the NIC drivers. Whenever probes are added, removed or relocated, the kernel must be recompiled and rebooted. A kernel configuration variable has been added so that probe macros can be conditionally compiled "in" or "out" of the kernel, so it is easy to create production and test versions of the kernel.

One of the parameters supplied to every probe macro is a "keymask", which is a 32-bit mask containing one or more bits set to 1. All probes having the same bit set to 1 in their keymask form a set, and these key sets can be enabled or disabled by user-level programs through the use of appropriate system calls. For example, when instrumenting the TCP-IP protocol stack, probes placed in the socket interface are given a keymask that is different from that used by probes in the TCP layer, which differs from the keymask used by probes in the IP layer, etc. This gives the user independent control of the probes at each layer. When probes are "off", they introduce very little overhead, essentially just the cost of a test and branch.

The timings recorded by a probe are those given by the Intel Pentium's time-stamp counter [2]. This counter is automatically incremented by the processor on every hardware clock cycle, and provides FKT with a very precise, real-time accounting of every CPU cycle without any additional hardware.

The data recorded by a probe consists of the time at which execution flows into the probe, as given by the processor's real-time cycle clock, a unique identification code assigned to the probe, the pid of the current process, the number of the processor (on multi-processors) and any additional parameters given by the programmer. The minimum amount of data recorded by a probe with no additional parameters is 12 bytes.

Probes record data into a large kernel buffer. In the present implementation, probing is automatically suspended when the buffer fills, but work is underway to write this buffer to disk or over a network while probing is active. In any case, the buffer is intended to be large, on the order of half the available physical memory of the machine, and the amount of information recorded by each probe is small, so traces covering many tens of seconds of intense operating system activity can be recorded.

After a measurement session has finished, a user-level program writes the kernel probe buffer to a disk file, and a second user-level program analyzes this file when time and space constraints are not critical and will not perturb the measurements themselves. This analysis is under the control of the programmer and allows him or her to accurately account for every cycle of the CPU during the sampling interval.

2 DESIGN

The use of FKT consists of two phases: the measurement phase, and the analysis phase. The measurement phase requires the use of a user-level program, called `fkt_record`, that controls the kernel probes and writes the data recorded by the probes to a file. The analysis phase is another user-level program, called `fkt_print`, that analyzes the data file recorded in the first phase.

2.1 KERNEL COMPONENTS

The kernel mechanism consists of three components:

1. a set of macros that can be inserted in existing kernel functions. The form of these macros is:

   FKT_PROBEx(KEYMASK, CODE, ...)

   where x is a digit in the range 0..5 that indicates the number of parameters whose values replace the ... in the macro. Each macro generates a test of the KEYMASK against the currently enabled key set value stored in the kernel variable `fkt_active`. Those that match trigger a call to an appropriate kernel function.

2. a set of kernel functions invoked by the macros that record into a large kernel buffer the information provided by the macro along with the current process id, the processor number and the processor cycle clock. The principle function is given by the prototype

   void fkt_header(unsigned int head, ...)

   Here "head" is a value constructed by the macro

   $$((CODE << 8) + (3 + x) \times 4)$$

   where x is the number of parameters whose values replace the "..." in the call. Because most calls involve only zero or one parameter, there are two optimized functions for these special cases:

   void fkt_eax0(void)

   and

   void fkt_eax1(void)

   where the parameters are passed via registers, a fast, non-standard technique.

3. a set of new kernel functions that set up, control, and preserve the results of the recording. These functions are invoked by system calls from user-level programs.

   int fkt_setup(unsigned int keymask)

   dynamically allocates a very large set of contiguous pages in the kernel as the probe recording buffer. It also sets up a number of global variables in the kernel that are used by the probing functions to control access to this buffer. For example, the global variable `fkt_active`, which contains the key set used to determine which probes are active and which are not, is set to the value of the `keymask` parameter. In addition, `fkt_setup` initializes the buffer with some information that enables the off-line analyzer to calibrate the cost of a single probe (80 cycles) and to map the processor's real-time cycle clock values onto the wall-clock time of day.

   int fkt_keychange(int how, unsigned int mask)

   dynamically changes the kernel's `fkt_active` keymask, thereby controlling which probes are active and which are not.

   int fkt_getcopy(int maxints, unsigned int *buffer)

   copies the kernel's recording buffer into the user-level "buffer" from which the user-level program can write it to disk.

   int fkt_reset(unsigned int keymask)

   is the same as `fkt_setup` except that it reuses an existing kernel buffer rather than allocating a new one. This enables several measurement sessions to be performed between an initial call to `fkt_setup` and a final call to `fkt_endup`.

   int fkt_endup(void)

   dynamically frees the kernel's recording buffer at the end of a measurement session.

2.2 USER-LEVEL PROGRAMS

In addition to the kernel components, there are two user-level programs that provide convenient control of the tracing mechanism.

2.2.1 FKT_RECORD

The program `fkt_record` can be used to probe any arbitrary user "target" program which is specified as a parameter to `fkt_record`. It operates by first calling `fkt_setup` to allocate the kernel buffer and to initialize the kernel's `fkt_active` keymask to a value given as a command-line option to `fkt_record`. It then forks a child process which in turn calls `exec` to run the specified user "target" program. When this program completes, `fkt_record` disables all probes, copies the kernel buffer containing the trace data to a file, and finally calls `fkt_endup` to deallocate the kernel buffer.

2.2.2 FKT_PRINT

This program analyzes the trace data file off-line to give various evaluations, such as the observed flows and nestings, the time spent in various functions, and an exact accounting for every processor cycle during the measurement interval. Although `fkt_print` offers a "standard" set of possible analyses as selectable options, users are free to modify this program or to write their own analyzer to study the trace data in a manner most relevant to their interests.

3 NETWORK ANALYSIS

Much has been written about various ways to evaluate networking protocols, especially TCP [3][1].

FKT has been used to investigate the TCP, UDP, IP, and LAMP[4] protocol stacks in Linux. Probes were inserted at the entry and exit points for all the major kernel functions in these stacks, and then two user-level test programs were run: a ping-pong program to test round-trip latency, and a blast program to test throughput. The results enable users to evaluate the effects of caching, interrupt generation, onboard checksum calculation, processor speed, network bandwidth and network connectivity on program performance. The margin of error for the timings of a typical routine is less than 1% with 95% confidence.

An interesting measurement was made to determine the effect of computing the TCP checksum in software versus on the network interface card (NIC). These tests were run on two 600 MHZ Pentium III processors with 256 MB of memory interfaced to 10 Mbps Ethernet by 3Com "Boomerang" cards. The `ping-pong` program was run for 1,000 messages of 5,000 bytes each. Figure 1 shows the results on the client side with hardware checksumming, and Figure 2 shows the results of the same test on the same NIC, but with the checksum calculated in software. In these Figures, only those functions involved in the receiving side of the TCP-IP stack are shown. The column labeled "Cycles" is the total number of processor cycles spent in the function, "Count" is the total number of times that function was performed, "Average" is "Cycles" divided by "Count", and "Time" is "Average" converted to microseconds. The column labeled "Percent" is "Cycles" relative to the total time taken in the test, and "Relative" is "Cycles" relative to the total time excluding time spent in the idle process (which was about 94% of the overall total) and other unrelated processes.

The kernel function that calculates the software checksum is `tcp_v4_rcv_csum`, and a comparison of the data for this function in the two figures shows that the percentage of time spent in this function increased from 767,205 cycles (.38% of the total time) for hardware checksumming to 21,332,718 cycles (10.50%) for software checksumming. This is a net gain of 20,565,513 cycles when the checksum calculation is pushed into the hardware.

However, we don't see this gain when we look at the total net gain of the test runs, which is only 7,777,731 cycles, less than 38% of the improvement due to the hardware checksum calculation.

The reason for this can be seen by looking at the timings for the function `tcp_recvmsg_ok`, which is the routine that copies the TCP message payload from kernel to user space. The time spent in this routine was only 5,076,866 cycles (2.50%) when the checksum was calculated in software, but increases dramatically to 17,659,206 cycles (8.83%) when the checksum was calculated on the NIC. This is a loss of 12,582,340 cycles in this routine alone, so that together these two routines produce a net gain of only 7,983,173 cycles when checksum calculation is moved from software to hardware. This gain is within 115,442 cycles (less than 1.5%) of the total net gain for all the receiving functions. Therefore, these two routines are clearly the major contributors to the observed difference in total performance between hardware and software checksumming. (This is confirmed by comparing the other entries in the two Figures – only the lines for these two functions vary by more than a fraction of a percent.)

We concluded, and further measurements verified, that the unexpected loss of cycles in `tcp_recvmsg_ok` was due to the effect of the processor cache. When the NIC calculates the checksum, the message is read into memory from the NIC via a DMA operation but is not loaded into the processor cache until `tcp_recvmsg_ok` copies it from kernel to user space. During this copy, the processor encounters a "stall" while waiting for the data to be fetched from kernel memory, and the cache is loaded with the same data twice (once at the kernel address, once at the user address).

Functions for receiving data:

Name	Code	Cycles	Count	Average	Time	Percent	Relative
sys_socketcall	66	2137918	4000	534.00	0.89	0.05%	1.07%
vortex_interrupt	32a	12061070	6003	2009.00	3.36	0.30%	6.03%
tcp_recvmsg_sched	3ab	6256616	8994	695.64	1.16	0.16%	3.13%
boomerang_rx	32b	13206055	6001	2200.64	3.68	0.33%	6.60%
tcp_v4_rcv_csum	331	767205	6000	127.87	0.21	0.02%	0.38%
tcp_ack	32f	6228779	3000	2076.26	3.47	0.16%	3.11%
tcp_data	32e	652808	1002	651.50	1.09	0.02%	0.33%
tcp_rcv_established	32d	8033440	6000	1338.91	2.24	0.20%	4.02%
tcp_v4_rcv	3a9	4298719	6000	716.45	1.20	0.11%	2.15%
ip_local_deliver	343	2633466	6000	438.91	0.73	0.07%	1.32%
ip_rcv	3a8	3498711	6000	583.12	0.97	0.09%	1.75%
net_bh	3b0	3652097	6001	608.58	1.02	0.09%	1.83%
tcp_recvmsg_ok	332	17659206	4000	4414.80	7.38	0.45%	8.83%
tcp_recvmsg	330	10791950	4000	2697.99	4.51	0.27%	5.39%
inet_recvmsg	3a7	1793485	4000	448.37	0.75	0.05%	0.90%
recvfrom	387	3091758	4000	772.94	1.29	0.08%	1.55%
Total		96763283				2.37%	48.36%

FIGURE 1: Summary of TCP Receiving Functions with Hardware Checksumming

Functions for receiving data:

Name	Code	Cycles	Count	Average	Time	Percent	Relative
sys_socketcall	66	2131546	4000	532.00	0.89	0.06%	1.05%
vortex_interrupt	32a	12061827	6004	2008.00	3.35	0.36%	5.94%
tcp_recvmsg_sched	3ab	6609515	8997	734.64	1.23	0.20%	3.25%
boomerang_rx	32b	13235330	6001	2205.52	3.69	0.39%	6.52%
tcp_v4_rcv_csum	331	21332718	6000	3555.45	5.94	0.63%	10.50%
tcp_ack	32f	6307859	3000	2102.62	3.51	0.19%	3.11%
tcp_data	32e	652096	1002	650.79	1.09	0.02%	0.32%
tcp_rcv_established	32d	8247336	6000	1374.56	2.30	0.24%	4.06%
tcp_v4_rcv	3a9	3895111	6000	649.19	1.08	0.12%	1.92%
ip_local_deliver	343	2641966	6001	440.25	0.74	0.08%	1.30%
ip_rcv	3a8	3542009	6001	590.24	0.99	0.10%	1.74%
net_bh	3b0	3703918	6001	617.22	1.03	0.11%	1.82%
tcp_recvmsg_ok	332	5076866	4000	1269.22	2.12	0.15%	2.50%
tcp_recvmsg	330	10336863	4000	2584.22	4.32	0.31%	5.09%
inet_recvmsg	3a7	1791463	4000	447.87	0.75	0.05%	0.88%
recvfrom	387	2972128	4000	743.03	1.24	0.09%	1.46%
udp_rcv	33f	2463	1	2463.00	4.12	0.00%	0.00%
Total		104541014				3.01%	51.47%

FIGURE 2: Summary of TCP Receiving Functions with Software Checksumming

When the checksum is not calculated in hardware, the software routine `tcp_v4_rcv_csum` that does calculate it encounters a processor stall and loads the cache once as it fetches the message data from kernel memory. When the data is later copied from kernel to user space by `tcp_recvmsg_ok`, it is already in the cache at the kernel address, so there is no additional processor stall, and the data only has to be entered into the cache once again (at the user address).

The number of processor stalls and the number of cache entries is identical in both cases, so what these measurements show is the unexpected result that loading the cache in two separate operations (a read followed much later by a write) is actually significantly faster than loading it twice (a read followed immediately by a write) on the same operation. The practical consequence of this is to significantly reduce the speedup that would otherwise be obtainable when the TCP checksum is calculated in hardware rather than software.

In another example of the utility of FKT, we discovered from our traces that the NIC for one interface was generating significantly more receive interrupts than that for a different interface. This led us to look at the algorithms used by the two drivers, and to change one of them to reduce the number of interrupts generated, thus reducing the overhead and increasing network performance on that NIC.

We also noticed an unexpected "improvement" in our kernel's implementation of TCP. By looking at the parameter values produced by our probes for a particular TCP routine we found that the maximum payload size was consistently 12 bytes smaller than expected. We quickly determined from the source code for that routine that this version of TCP was using the new "PAWS" timestamp option on all packets. Although we could have determined this by wading through all the TCP source code, or by analyzing the output of a network sniffer, our FKT tracing made it obvious that something was different, and also pinpointed the exact routine containing the change. This is clearly of huge importance to users with time-critical applications who need to know what, if anything, changed when they receive a new kernel release.

As a final example, we observed that the round-trip latency we were achieving from our Gigabit Ethernet was much lower than could be expected from the raw bandwidth of the network and the processing time taken by the TCP/IP protocol stack. Analysis of the FKT trace data allowed us to prove that the time taken by the firmware on the Gigabit Ethernet NIC was substantial, much more than on Fast Ethernet NICs for example, and that this was a major cause of the poor performance.

4 OTHER USES OF FKT

In addition to the use of FKT for performance evaluation, two other uses have proven to be equally valuable in practice: debugging and instruction.

4.1 DEBUGGING

FKT is useful for kernel debugging because the extra parameters supplied to the probe macros can be anything the user desires. It is therefore possible to use this tool to "dump" kernel values at arbitrary places in the kernel code, and these dumps can be selectively turned on and off while the kernel is running. Furthermore, the macros are all designed so that they can be compiled under two different configurations – one is the "full-parameter" configuration, which is what has been described above. The second is the "no-parameter" configuration, in which all the extra parameters are ignored. Thus, once the kernel has been debugged and the debugging facility is no longer needed, all that needs to be done is a simple recompilation, and the debugging aspects of the macros disappear, leaving just the performance evaluation aspects. Although a separate set of debugging macros could obviously supply the same information and be configured in the same way, the point is that a separate tool is NOT needed, and separate insertions into the kernel code are NOT needed.

4.2 INSTRUCTION

The other unexpectedly useful task to which FKT can be put is education. When reading source code it can be a daunting task to figure out what code is executed when, especially with the type of coding found in the kernel (which is accompanied by very few comments). Clearly, a trace of the running code supplies exactly that information, and when accompanied by the extra values of the macros' parameters, the trace forms a valuable aid to learning about the kernel's operation. In the case of the Linux kernel this is even more valuable, because much of the flow is parameterized through the use of "jump tables" or "transfer vectors". While this is an excellent technique for writing general, modular code, it makes it impossible to determine from looking at the source code where a transfer through a jump table ends up. The document produced by a trace answers that question succinctly.

5 RELATED WORK

Many tools [5, 6, 7, 8, 9] have been developed to measure kernel performance. The DIGITAL Continuous Profiling Infrastructure (DCPI) [8] uses periodic sampling of hardware performance counters in the DEC Alpha, and infers times spent in functions by sampling techniques. This requires no kernel modifications except the introduction of a special driver. FKT on the other hand requires insertion of probe macros into existing kernel functions, but is able to produce precise, fine-grained traces as well as total times in functions.

Both KernInst [9] and Kitrace [7] require dynamic on-line modification of a running kernel, a very dangerous operation that requires intimate knowledge of the kernel code. FKT requires off-line modification of kernel source, a much safer alternative because the person making the changes can see the source context of the inserted probe and can obtain the benefit of the compiler-time check after the insertion. Furthermore, the FKT modifications become a permanent part of the kernel and are not lost when the kernel crashes or is rebooted. The cost of modifying kernel source code, recompiling and rebooting is so short in Linux, especially when loadable modules are used, that this approach can be considered "quasi-interactive".

6 CONCLUSION

This paper presents an overview of the design of FKT, a Fast Kernel Tracing facility which enables kernel programmers to obtain precise, time-stamped traces of the dynamic activities of kernel-level code. Each probe records a small amount of data into a kernel buffer that is later analyzed off-line. This recording is performed with very low overhead in order to minimize the perturbation of the running kernel. Yet the data provided is very precise and fine-grained, which permits detailed accounting of every processor cycle. Even just the basic kernel trace consisting of all system calls, interrupts, traps and task switches provides a good overview of program behavior with no extra effort on the part of the user.

An example was presented that demonstrated the utility of FKT in understanding the performance gains and losses when TCP checksumming was performed by the network interface card versus in software. We demonstrated that an unforeseen cache effect limited the performance increase to less than 38% of what could be expected from examination of the NIC performance alone.

Besides its obvious merits as a tool for obtaining very high-precision performance data, FKT has also proven to be a valuable aid to debugging and educational instruction, since it can give a precise picture of the exact flow taken by the processor and can show the values of variables at strategic points along the way.

7 ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation Grant 9601602.

REFERENCES

[1] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[2] Intel. *Intel Architecture Sofware Developer's Manual, Volume 2: Instruction Set Reference, Order Number 243191*, 1999.

[3] M. Allman and A. Falk. On the effective evaluation of tcp. *Computer Communication Review*, 29(5):59–70, October 1999.

[4] Sadhna Ahuja and Robert Russell. A high-performance transport protocol for local area networks. In *Proceedings of the 18th IASTED International Conference on Applied Informatics*, February 2000.

[5] John K. Ousterhout, Herve 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. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24. ACM, December 1985.

[6] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Sherriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–211. ACM, October 1991.

[7] Geoffrey H. Kuenning. Kitrace: Precise interactive measurement of operating systems kernels. *Software – Practice and Experience*, 25(1):1–21, January 1995.

[8] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Chemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 1–14. ACM, October 1997.

[9] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 117–130. ACM, 1999.