The Use of Microcode Instrumentation for Development, Debugging and Tuning of Operating System Kernels

Stephen W. Melvin
Yale N. Patt

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

We have developed a tool based on microcode modifications to a VAX 8600 which allows a wide variety of operating system measurements to be taken with minimal perturbation and without the need to modify any operating system software. A trace of interrupts, exceptions, system calls and context switches is generated as a side-effect to normal execution. In this paper we describe the tool we have developed and present some results we have gathered under both UNIX 4.3 BSD and VAX/VMS V4.5. We compare the process fork behavior of two different command shells under UNIX, look at context switch rates for interactive and batch workloads and generate a histogram for network interrupt service time.

1. Introduction

Two basic approaches to performance analysis are modeling and data collection. While modeling has important applications, it can become intractable for complex systems unless overly simplistic assumptions are made. The interactions that occur on a large multiprogrammed computer with a modern operating system are so varied and complex that empirical measurements are essential. Furthermore, it is important to carefully consider the quality of the data being collected and the cost of collecting it.

There are three fundamental approaches to data collection: hardware, software and microcode. Hardware monitors can take measurements with virtually no effect on the system. However, they can be inflexible, limited in resources, expensive and cumbersome to operate. Alternatively, the operating system kernel can be modified to collect data which, although satisfactory for some types of measurements, can be impractical and/or cause an unacceptable perturbation of the measurement for others. In addition, modifying the kernels of most operating systems is not a matter to be taken lightly. It is often difficult to determine the effect that a seemingly small change can have on other parts of the kernel.

Most of the advantages of both hardware and software methods can be achieved with a microprogrammed measurement gathering technique. By modifying the microcode, measurements can be taken with a very small effect on the system. In addition, microcode-based systems can be flexible and easy to use. Once the core microcode is installed that implements the data collection tool, everything else can be under software control. Furthermore, since the data collection takes place below the operating system there no need to modify the kernel and the same measurement can be taken on different operating systems.

The idea of using microcode to gather measurements has been around for a long time. The earliest mention of which we are aware was by Halbach in 1971 [5]. In this two page note, Halbach discusses the gathering of trace information through the use of microcode modifications. Armbruster discusses the gathering of instruction traces in [2] and Grätsh and Kästner provide a brief history of firmware monitoring in [4]. Chroust, Kreuzer and Stadler discuss a microprogrammed page-fault monitor in [3] and Agarwal, Sites and Horowitz discuss a microcode-assisted address tracer in [1].

We have implemented a microcode based event tracer which we call SPAM (for System Performance Analysis using Microcode) which differs from these previous methods in two important ways. First, it is specifically geared toward gathering real-time sensitive data. The quantity and type of information being gathered is carefully chosen to minimize perturbation of the system. Second, SPAM is a general facility that allows a wide variety of data to be collected rather than a specific tool to collect one particular type of data. We have also implemented an interface to the microcode to allow SPAM to be used without the need for microcode expertise or specific hardware knowledge.

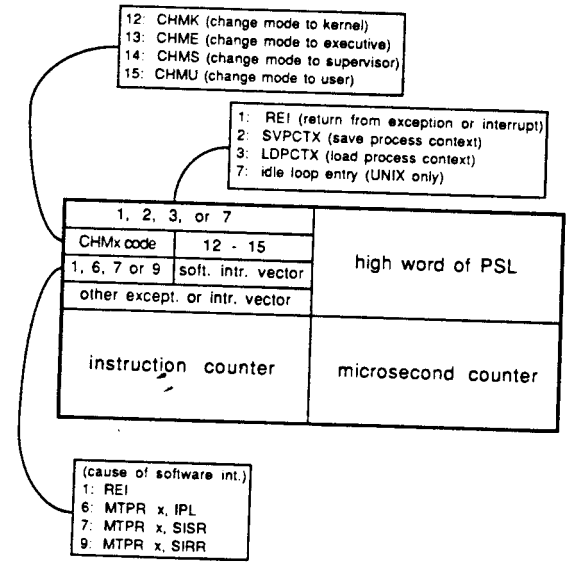
SPAM is based on microcode modifications to a VAX 8600 which include additional machine level in-

structions as well as side effects to existing microcode flows. It traces interrupts, exceptions, system calls and context switches. The information recorded for each event includes an instruction count, a microsecond count and the current mode of execution of the processor. Because the VAX architecture is preserved, all operating system functions and utilities can operate without modification and because the overall perturbation is less than 10%, all normal system activities can be accommodated.

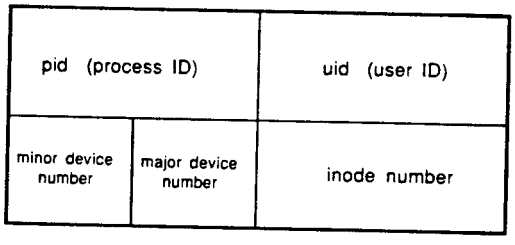
In previous papers [6],[7], we have discussed low level implementation details of SPAM. In this paper we concentrate on applications of this tool and provide several examples. We compare two different UNIX command shells with regard to how they fork processes, generate distributions for context switch rate for two different workloads and look at network interrupt service time. This paper is divided into four sections. Section 2 illustrates the interface to SPAM. The manipulation of the tool as well as the trace format are discussed. Section 3 presents the results we have collected and section 4 offers some concluding remarks and details of future work.

Figure 1
Record Formats:

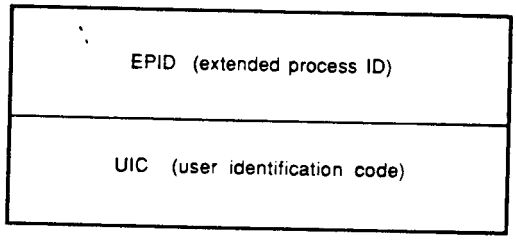
Format of basic record (8 bytes):



Format of LDPCTX extended record (8 bytes) - UNIX:



Format of LDPCTX extended record (8 bytes) - VMS:



2. The SPAM Interface

In this section we describe how SPAM looks to higher level programs. First, we discuss the basic processes involved in collecting data. Then we describe the format of the event record and provide an example of a small piece of a trace.

2.1. Trace generation, retrieval and analysis

The trace of events that SPAM generates is recorded in a trace buffer. This buffer is a reserved portion of physical memory the size of which can be varied in integer multiples of one megabyte. The buffer is implemented as a circular queue, allowing reads and writes to take place simultaneously. If the buffer overflows, tracing will automatically be disabled (it can also be explicitly disabled). A larger buffer will take more physical memory away from the operating system and thus perturb the measurement more, but may allow a more complete measurement to be taken by allowing a longer collection time. Depending on the workload, SPAM accumulates data at a rate of one half to two megabytes per minute. (VAX 8600's can be configured with up to 68 megabytes and we have taken measurements with buffer sizes ranging from 1 to 36 megabytes.)

There are three separate activities that take place in the use of SPAM: the generation of the event trace, the retrieval of the trace from the trace buffer, and the analysis of the collected data. The retrieval and analysis can either take place during the trace generation or afterward. If the minimum system perturbation is desired and the buffer can hold the size of trace desired, data collection and retrieval would be separate activities. The trace would accumulate in the buffer during the measurement period and would be retrieved afterward.

In certain circumstances, it may be desirable to retrieve and possibly analyze the data as it is being generated. This would be the case if the buffer is too small to hold the size of trace desired, and the effect of retrieving the data on the data being collected is insignificant. In this case, the trace could either be accumulated on disk or it could be analyzed dynamically (e.g. a histogram could be generated dynamically for a particular measurement). The advantage of writing the

data directly to disk is that then the post-processing can gather whatever data is desired. The advantages of doing the analysis in memory is that data can be collected over longer periods of time and perturbation of the system is less. A large amount of disk space may be required if traces are desired over a long period of time, however the data could be significantly compacted before being written to disk.

2.2. Record format

If tracing is enabled and the trace buffer is not full, an event causes an eight-byte record to be recorded. If that event happens to be a context switch, an additional eight-byte record is written to the buffer. This additional record is used to identify the new process.

The basic eight-byte record consists of a 16-bit microsecond count, a 16-bit instruction count, the high 16 bits of the processor status longword (PSL), and information identifying the event (see figure 1).

The instruction count field and the microsecond count field represent counters that are incremented once per instruction and microsecond respectively. We are not worried about overflow on the microsecond counter because timer interrupts occur every 10ms (10,000 counts) and we are not worried about overflow on the instruction counter because the processor is not fast enough to execute more than 65535 instructions in 10ms.

The most significant 16-bits of the PSL contains three pieces of useful information: the IS bit (indicating if the processor is currently executing on the interrupt stack), the current privilege level (kernel, executive, supervisor or user) and the interrupt priority level, or IPL. When the CPU is executing in kernel, executive, supervisor or user mode it is executing on behalf of a particular process, or in "process context". When executing on the interrupt stack, the CPU is in "system context", and, as far the the CPU is concerned, the activity is not associated with any particular process. (UNIX only uses kernel and user modes while VMS uses all four modes.)

The IPL is a five bit field that indicates the current interrupt priority level. Software interrupts take place at interrupt priority levels 1 to 15 and hardware interrupts take place at levels 16 to 31. The basic trace record also provides 16 bits of information to identify the associated event. In the case of system calls, which are realized by the VAX CHMK, CHME, CHMS and CHMU instructions, the type of system call is identified. In the case of software interrupts, the cause of the software interrupt is identified. Also traced are entries in to the "idle loop" in the case of a UNIX implementation. This is necessary in order to be able to identify idle time in contrast to VMS, which has a separate process for idle time.

Basic trace records for LDPCTX events are followed by an extended trace records. At the end of a VAX LDPCTX instruction, the CPU is executing in the context of a new process. The extended record provides information to identify this process. For UNIX, the pid and uid are simply the process ID and user ID of that process respectively. The major and minor device numbers along with the inode number uniquely identify the file which contains the executable image. For VMS the EPID is the extended process ID of the process and the UIC is the user identification code of the user associated with the process.

Figure 2
SPAM Trace Example

time	instr	mode	ipl	event
21,	2,	K,	31,	LDPCTX, pid = 5217, uid = 2192
9,	5,	K,	00,	REI
106,	136,	U,	00,	REI
12,	3,	I,	24,	timer interrupt
67,	102,	U,	00,	REI
9933,	32348,	I,	24,	timer interrupt
46,	101,	U,	00,	REI
679,	2014,	I,	21,	vector = 14c
245,	231,	I,	12,	REI, soft. lvl. 12
1182,	1728,	U,	00,	REI
102,	304,	I,	21,	vector = 14c
72,	74,	U,	00,	REI
1659,	5290,	I,	21,	vector = 14c
70,	88,	U,	00,	REI
5945,	19366,	I,	24,	timer interrupt
153,	223,	K,	02,	REI, soft. lvl. 2
58,	62,	I,	31,	SVPCTX

2.3. Sample trace

Figure 2 shows a small piece of an event trace that is helpful for understanding the level at which SPAM records information. It illustrates the activity between the load process and save process instructions for a compute bound process. Each line in figure 2 represents one basic record. The first column is the number of microseconds which have accumulated since the previous event. The second column is the number of instructions which have accumulated. The next two columns represent the mode and interrupt priority level of the CPU at the *end* of the applicable event. An **I** means the CPU was on the interrupt stack, and **K** and **U** refer to kernel and user modes respectively. The rightmost column indicates the particular event for each record. REI is the VAX instruction which returns from all exceptions and interrupts.

Consider the eighth record illustrated. This is the record for a hardware interrupt with a vector of 14c (hexadecimal). The trace indicates that after 245 microseconds and 231 instructions in the interrupt service routine a REI instruction is encountered. A level 12 software interrupt is immediately initiated. Thus, the hardware interrupt must have enabled the level 12 software interrupt. This routine (which happens to be the network interrupt service routine) took 1182 microseconds and 1728 instructions. Then, another REI was encountered which returned the processor to user mode; and so forth.

3. Results and Analysis

In this section, we present three different experiments we have performed with SPAM and analyze the results. These experiments are a comparison of the

process fork behavior of two different UNIX command shells, a context switch rate summary and a network interrupt service time summary. But before we discuss these experiments, we present a comparison between the information that the UNIX **time** command provides and the same information provided by SPAM. This was done to get an idea of the difference between software sampling and microcode accounting methods.

3.1. A comparison of *time* and SPAM

In order to determine the amount of CPU time a particular command uses, a builtin command is provided in UNIX called **time** which reports usage statistics. If the following command is typed, for example:

```
% time cc *.c
```

the C compiler will be invoked on all source code (.c) files in the current directory and after it is done, a line will be displayed with a usage summary. We ran this command on the source code for a hardware simulator of approximately 14,000 lines. The code consisted of 9 source files (.c) and 5 header files (.h) and there was no other user activity at the time. After execution, the following information was displayed:

```
88.6u 4.3s 1:40 92% ...
```

In this section we will only concentrate on the time information, provided in the first three fields. These tell us that approximately 88.6 seconds of user time and 4.3 seconds of system time were used over an elapsed time of 100 seconds. The results of **time** show only a vague picture of what happened. The information is based on a sampling of the current CPU state during every hardware interrupt, and thus represents only an approximation to actual activity. However, hardware interrupts occur every 10ms, so over a period of 100 seconds, a total of about 10,000 samples were taken. We would therefore expect **time** to be fairly accurate in this example.

Unlike the sampling technique, SPAM captures all events and exhaustively accounts for all time. During the same compilation that generated the data above, SPAM was collecting its data in the trace buffer. After completion of the compilation, the trace buffer was retrieved and analyzed. By examining the SPAM data, we can get a clearer idea of the sequence of events (see figure 3).

We see from the SPAM results that this particular C compilation involved 29 processes in all. Each source code file caused three processes to be invoked, /lib/cpp (the preprocessor), /lib/ccom (the main part of the compiler) and /bin/as (the assembler), in that order. The other two processes were /bin/cc (which was the first process created and which created all the other processes) and /bin/ld, the linking loader (which was the last process created). Figure 3 shows the number of microseconds accumulated for each process in user and kernel modes. Totals are also provided for each of the 5 executable files involved. The elapsed time was calculated from the **vfork** system call which invoked the first process until the **exit** system call for the last process. SPAM accumulated for a process all the time that the process was loaded and the CPU was executing in process context.

We can see that the numbers are relatively close (88.6 seconds vs. 87.600 seconds, 4.3 seconds vs. 4.2411 seconds and 100 seconds vs. 100.371 seconds). Note that SPAM provides 7 significant digits while **time** provides 2 to 3. The results in figure 3 for SPAM are only an example of the information available. Other interesting information that could have been collected would be number of page faults, number of context switches, a breakdown of system calls (which processes executed which ones, how often and how long they took), etc. The primary purpose of this experiment, however, was

Figure 3
A Comparison of "time" and SPAM

```
Total microseconds elapsed:      100370846

                 user mode:       87600291      (87.28%)
               system mode:        4241052      ( 4.23%)

                idle loop:        7201685      ( 7.18%)
          interrupt stack:        1323347      ( 1.32%)
          other processes:           4471      ( 0.00%)
```

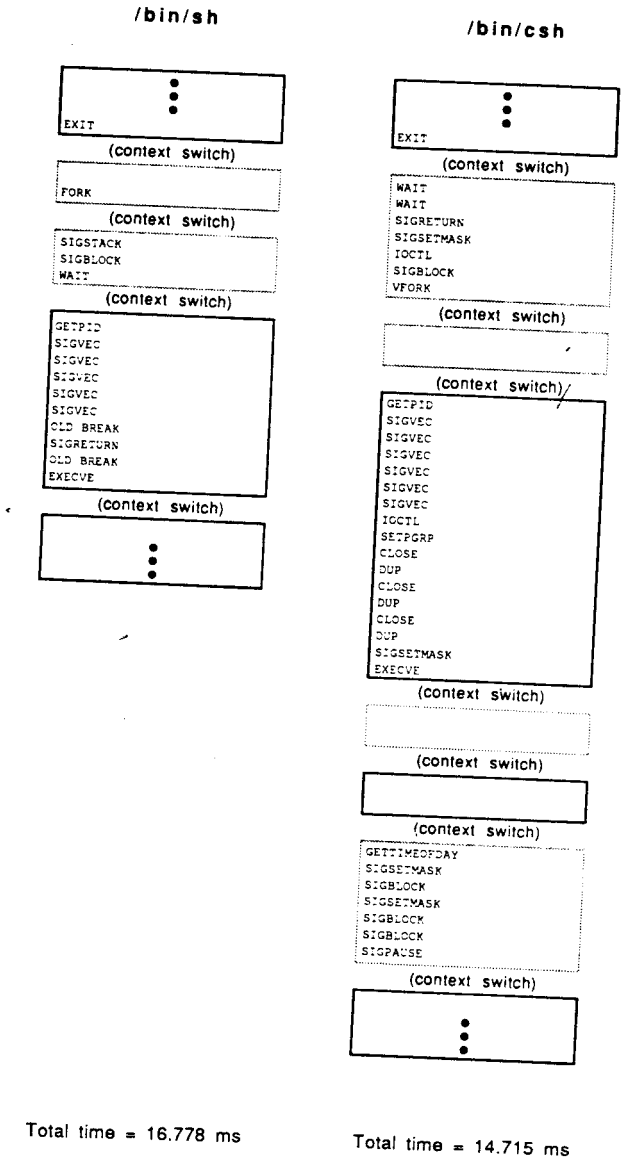
Breakdown by process:

pid	text file	user	system
29	/bin/cc:	14867	152611
30	/lib/cpp:	318043	83086
31	/lib/ccom:	4262900	140122
32	/bin/as:	1545009	162741
33	/lib/cpp:	394557	90895
34	/lib/ccom:	5896387	160751
35	/bin/as:	1988277	157042
36	/lib/cpp:	625445	99728
37	/lib/ccom:	4876329	122970
38	/bin/as:	1625985	145852
39	/lib/cpp:	139763	49582
40	/lib/ccom:	597342	95564
41	/bin/as:	159550	68163
42	/lib/cpp:	631245	98537
43	/lib/ccom:	8972349	160511
44	/bin/as:	4532909	307682
45	/lib/cpp:	262327	61595
46	/lib/ccom:	3082209	120642
47	/bin/as:	1028389	118835
48	/lib/cpp:	1071737	136719
49	/lib/ccom:	22353016	220471
50	/bin/as:	6697573	350350
51	/lib/cpp:	401926	79377
52	/lib/ccom:	7694953	144650
53	/bin/as:	2882762	197728
54	/lib/cpp:	431938	67698
55	/lib/ccom:	3142220	120465
56	/bin/as:	1235490	161363
57	/bin/ld:	734794	365322

Totals by text file (percentage of total execution):

text file	user		system	
/bin/cc:	14867	(0.02%)	152611	(0.17%)
/lib/cpp:	4276981	(4.66%)	767217	(0.84%)
/lib/ccom:	60877705	(66.29%)	1286146	(1.40%)
/bin/as:	21695944	(23.62%)	1669756	(1.82%)
/bin/ld:	734794	(0.80%)	365322	(0.40%)

Figure 4
Process Forking in Two UNIX Command Shells



to check the time obtained by SPAM against the time obtained by the operating system.

3.2. Process forking in UNIX

In this section we will look at two different UNIX methods for forking a process. A UNIX command shell creates processes to run each executable file. For example, if a command line of the following form is entered:

```
% a ; b
```

two processes will be created for each of the two executable files 'a' and 'b'. In this section we will take a look at the sequence of events and the amount of time required to recover from a previous process and start a new process.

In the above example, when the program 'a' is done, it executes an **exit** system call. This causes the parent shell process to wake up. This process then forks off a process to execute 'b' and goes back to sleep. In UNIX BSD, there are two different command shells in popular use: the Bourne shell, which we will refer to as "sh" and the C shell, which we will refer to as "csh".

We examined a SPAM trace for process forks for both shells and summarize the results in figure 4. This figure summarizes the sequence of events from the **exit** of the first process to the first user instruction executed in the new process. Shown in each box are the names of which system calls are used and in what sequence. One primary difference between csh and sh is the use of the optimized **vfork** system call as opposed to the **fork** system call. For **vfork**, the child temporarily "borrows" the parents process state since it only expects to need it long enough to create a new environment while for the **fork** system call, the state of the parent process is copied for the child. The total execution times indicate that the overhead of forking a process in csh is about 12% less than that for sh.

Note that while we have shown how SPAM can be useful for comparing two existing process forking techniques, it could also be used to evaluate new techniques for accomplishing the same task. This could be useful for the development of new techniques for process forking or tuning existing methods.

3.3. Context switch rate

In this section we examine the number of instructions between context switches for two different workloads: a synthetic workload under VMS and a real UNIX workload. The VMS measurements were taken on a VAX 8650 which was being driven by a second 8650 simulating 500 users running a debit/credit application. The SPAM-equipped 8650 had 64 megabytes of physical memory, of which we reserved 16 for the trace buffer. After the application had reached a steady state (all users logged in), SPAM was enabled and data was collected until the buffer was full. This took about 8.23 minutes, which translates to 1.94 megabytes per minute. During this period, 1.469 billion instructions were executed, which translates to 2.97 MIPS.

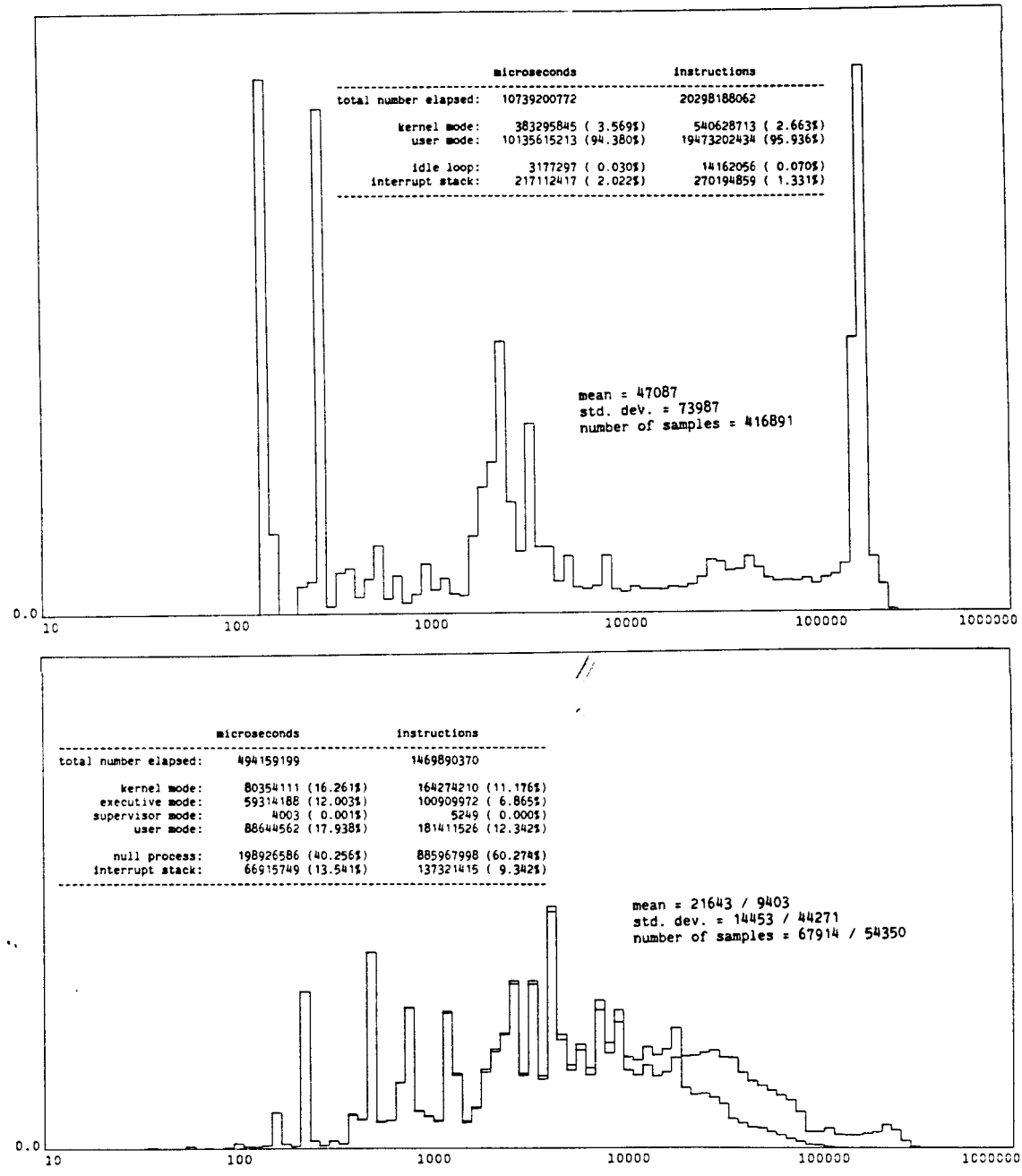
We gathered the number of instructions between context switches for this data set using two different schemes. In the first method, we simply counted the number of instructions between each context switches. In the second method, we discarded the context switches for the NULL process. (In VMS, a special process, the NULL process, is executed when there is nothing else to do.) The first measurement is what the hardware actually sees while the second measurement is useful for understanding the affect of idle time. A summary of the number of instructions and the number of microseconds spent in each of the five modes is presented in the lower half of figure 5 along with the distributions for the context switch rate. The lower curve and the second of the

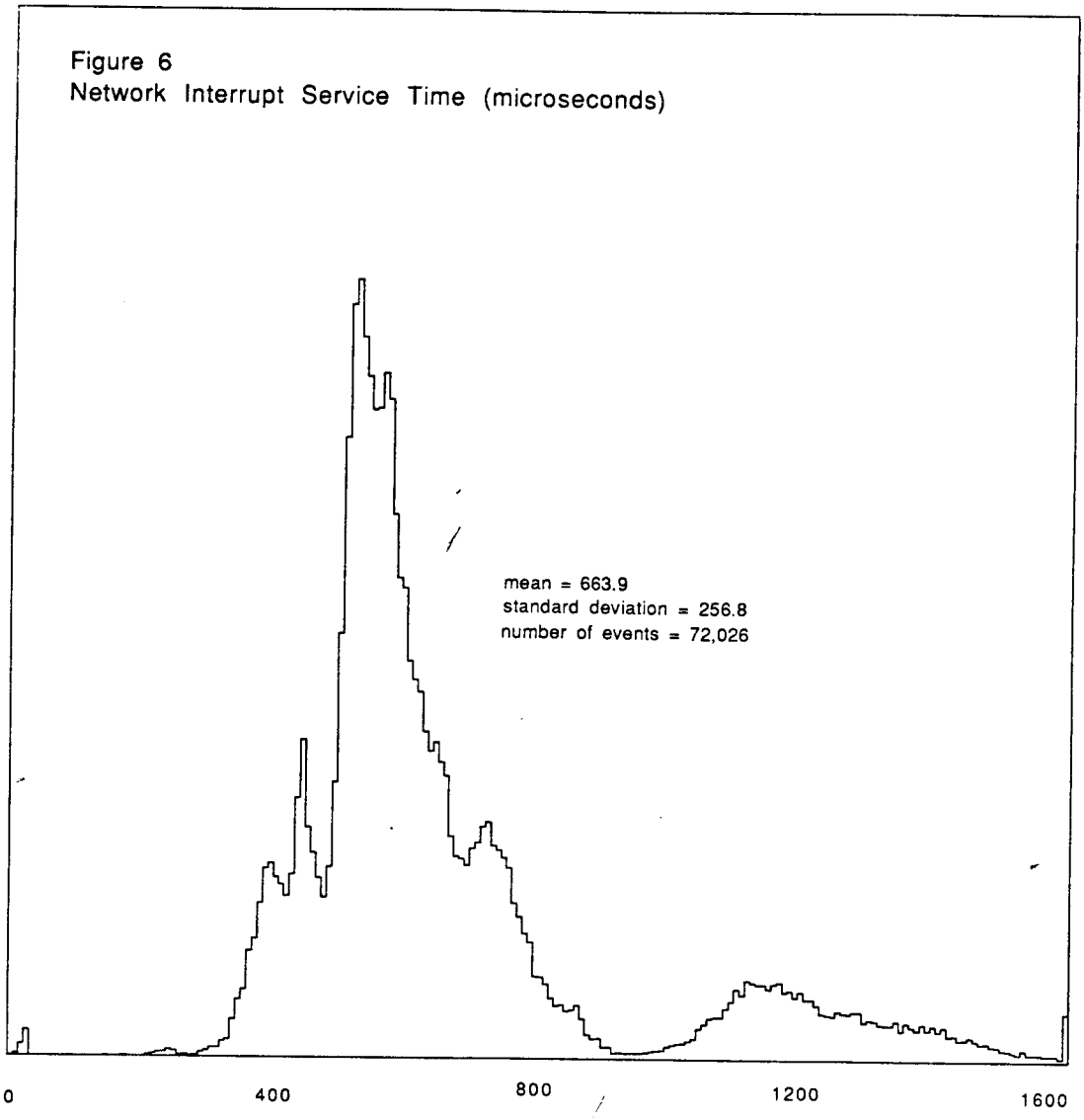
two numbers provided is for the case with the NULL process excluded.

The second workload which we used in this experiment was a more or less "typical" three hour period on a UNIX 4.3 BSD system. There were four to five users logged in running a variety of interactive jobs and two compute bound background jobs running which dominated the CPU activity. We were able to collect data over such a long period by running the data analysis process while the data was being generated. In this case, the buffer only needed to be 1 megabyte. We wrote the data collection program so that it would discard context switches for itself in order to reduce the perturbation of the measurement. However, this process accounted for only 3.22% of total number of instructions executed.

In the three hour period, 20.30 billion instructions were executed at an average rate of 1.89 MIPS. A summary of the number of instructions and the number of

Figure 5
Batch and Interactive Context Switch Rates





microseconds spent in each of the three modes is presented in the upper half of figure 5 along with the distributions for the context switch rate.

The two curves in figure 5 represent two very different sorts of activity. In the debit/credit workload (a transaction processing application) the environment is very interactive, but controlled. Note that the horizontal axes on these graphs are on a logarithmic scale. This curve is more highly skewed than might otherwise be apparent. More than 17% of the density of this curve is below 1000. The UNIX workload, on the other hand, is predominately batch dominated, but more variable. The average number of instructions between context switches in this case is almost 50,000. However, note the high peaks at low instruction counts. In this workload, more than 32% of the density is below 1000, almost twice the value for the previous curve. This activity is most likely due to screen editor behavior, where each time a character is typed, the associated process becomes runnable but doesn't have much to do. Thus, the heavy batch behavior pushes up the average number of instructions between context switches, but a closer look reveals a distribution even more highly skewed than the previous one.

3.4. Network communication overhead

A useful piece of information for the development and tuning of network communication protocols is the the overhead of sending a message across the network. That is, how much CPU time is spent in processing a network message. For UNIX this is separated into two functions: the interrupt service routine and the network server process.

The CPU responds to a network event by generating a software level 12 interrupt in response to the actual hardware interrupt. The network interrupt handler then does the absolute minimal amount of work necessary to get the message from the hardware buffer into an

operating system buffer. Then, a separate process works on behalf of the operating system to determines how to get that message to the proper destination process by executing the appropriate protocols. The time spent servicing the network interrupt in addition to the time used by the operating system process would give the total amount of overhead incurred by a network event. The network interrupt service time is incurred immediately upon receipt of a message and that the rest of the overhead can be incurred when the operating system determines it is convenient to do so.

We wrote a program to continuously read SPAM records as in the previous experiment and generate a histogram of the time spent in the network interrupt routine. This program used between 2% and 7% of the CPU, but it did not generate any network traffic, so the program itself didn't affect the measurement. Figure 6 shows the results from a particular data collection run. It was taken over approximately a 8 hour period during the middle of the day with moderate network activity. It shows that a network event causes the system to use about 0.6 milliseconds of CPU time. This type of curve could be useful in evaluating different schemes for handling network traffic.

4. Conclusions

In many environments, empirical measurements on computer systems are essential. Carrying out these measurements is often a difficult task at best, particularly when they involve operating system kernels. A choice is usually made between a hardware monitor or modifications to the software. We believe that a third choice, the use of microcoded instrumentation, results in a measurement system with significant advantages over other approaches and disadvantages which aren't as significant. Unlike a purely software method, the perturbation is smaller and modifications to the kernel are not required and unlike a hardware monitor, we have the flexibility, low cost and ease of use of a software system.

We have discussed a microcode based measurement tool we have developed, SPAM, and illustrated the type of data it can collect. SPAM emphasizes the collection of real-time sensitive data and it provides a very general data collection environment, removed from the low-level details of its implementation. The data which can be provided could be useful for development of new operating systems and new operating system features, the debugging of existing operating systems, and tuning of operating system functions (e.g. choosing those 'magic' numbers that operating systems seem to have so many of). We would like to point out, however, that we have only scratched the surface with this paper. Many improvements could be made to allow an even more useful tool.

The current record captures a core set of information that should be useful in most applications, but the need might also exist to trace measurement specific data, for example a particular memory location. In addition, trace triggering could be more general. The user should be able to specify exactly when tracing should be enabled and disabled (e.g. only certain processes or users, only after certain events have occurred, etc.) and also be able to control the types of events that are traced. We are also further developing the user interface such that people involved in operating system measurement can manipulate SPAM to gather a wide variety of measurements without the need to be familiar with details of its implementation.

5. Acknowledgement

The authors wish to acknowledge the Digital Equipment Corporation for their generous support of our research, in particular Bill Kania for providing us with the VAX 8600 in order to enhance our ability to do research in microarchitecture and microprogramming; also, Fernando Colon Osorio, Mario Troiani, Nii Quaynor, Steve Ching and Harold Hubschman, from DEC's High Performance Systems and Clusters Group in Marlboro. We would also like to acknowledge Joe Pasquale for many helpful suggestions. Our work in microarchitecture is part of a larger architectural research effort at Berkeley, the Aquarius Project. We acknowledge our colleagues in the Aquarius group for the stimulating environment they provide. Finally, we acknowledge that part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

REFERENCES

[1] Anant Agarwal, Richard L. Sites, and Mark Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 2-5, 1986, pp. 119-127.

[2] C. Edward Armbruster Jr., "A Microcoded Tool to Sample the Software Instruction Address," *Proceedings of the 12th Annual Workshop on Microprogramming*, 1979, pp. 68-72.

[3] G. Chroust, A. Kreuzer, and K. Stadler, "A Microprogrammed Page Fault Monitor," *Microprocessing and Microprogramming*, Vol. 8, 1981, pp. 247-256.

[4] Wolfgang Grätsch, and Horst Kästner, "Firmware Monitoring - History and Perpsective," *Microprocessing and Microprogramming*, Vol. 8, 1981, pp. 237-246.

[5] L. A. Halbach, "Microprogrammed Tracing Method," *IBM Technical Disclosure Bulletin*, Vol. 14, December, 1971, pp. 2164-2165.

[6] S. W. Melvin and Y. N. Patt, "A Microcode-Based Environment for Non-Invasive Performance Analysis," *Proceedings to The 19th Annual Workshop on Microprogramming*, October 15-17, 1986, New York, New York.

[7] S. W. Melvin and Y. N. Patt, "SPAM: A Microcode Based Tool for Tracing Operating System Events," *Proceedings to The 20th Annual Workshop on Microprogramming*, December 1-4, 1986, Colorado Springs, Colorado, pp. 168-171.