SPAM: A Microcode Based Tool for Tracing Operating System Events

Stephen W. Melvin
Yale N. Patt

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

ABSTRACT

We have developed a tool called SPAM (for System Performance Analysis using Microcode), based on microcode modifications to a VAX 8600, that traces operating system events as a side-effect to normal execution. This trace of interrupts, exceptions, system calls and context switches can then be processed to analyze operating system behavior for the purpose of debugging, tuning or development. SPAM allows measurements to be made on a fully operating UNIX system with little perturbation (typically less than 10%) and without the need for modifying the kernel.

1. Introduction

The interactions that occur on a large multiprogrammed computer with a modern operating system are so varied and complex that analyzing the behavior of the system is not a simple matter. Hardware monitors can be used to take measurements, and they don't affect what is being measured, but they are generally inflexible and have limited resources. In addition, hardware monitors are generally expensive and cumbersome to use.

Alternatively, the operating system kernel can be modified to collect data which, although satisfactory for some types of measurements, can be impractical or can 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 very 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 is no need to modify the kernel and the same measurement can be taken on different operating systems.

We have implemented a microcode based event tracer which we call SPAM (for System Performance Analysis using Microcode). It is based on microcode modifications to a VAX 8600 which include additional machine level instructions as well as side effects to existing microcode flows. It traces interrupts, exceptions, system calls and context switches and the information traced includes instruction counts, microsecond counts, processor mode and process and user IDs. Because the VAX architecture is preserved (with some minor exceptions), all operating system functions and utilities operate without modification. In a previous paper [1], we discussed the creation of a general environment for non-invasive performance measurement. SPAM is a specific tool that we have implemented within that general environment which is targeted at operating system measurements. This paper is divided into five sections. Section 2 illustrates SPAM as seen by the user. The format of the trace record is discussed along with how the tool is invoked. Section 3 provides an overview of the implementation of SPAM. Section 4 presents some results that we have collected. Finally, section 5 concludes with details of future work.

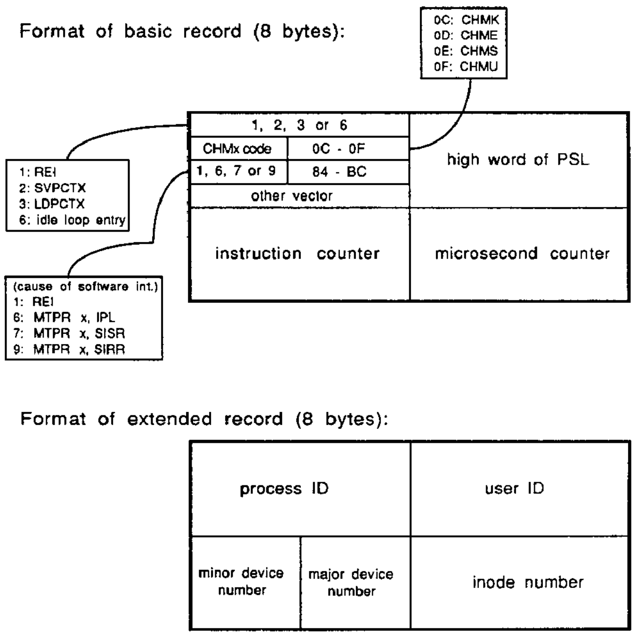
2. User level view of SPAM

2.1. Trace generation, retrieval and analysis

From the users' point of view, there are three separate activities taking place: the generation of the event trace, the retrieval of the trace from the trace buffer, and the analysis of the collected trace. The retrieval and analysis can either take place during the trace generation or afterward. If the least system perturbation is desired and the buffer can hold the size of the trace desired, data collection and retrieval would be separate activities. The trace would accumulate in the buffer (a reserved portion of physical memory, 1 megabyte in the current implementation) during the measurement period and would be retrieved afterward. Tracing can be explicitly disabled or it will automatically be disabled when the buffer is full.

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 the trace desired, and/or the effect 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. the generation of a histogram 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 the perturbation of the system is less. Depending on the workload, the current implementation of SPAM accumulates data at a rate of approximately one half of a megabyte per minute, so a large amount of disk space may be required if traces are desired over a long period of time. Note, however, that the data could be significantly compacted before being written to disk and future implementations of SPAM are likely to allow

Figure 1
Record Format



more selective event tracing, which could significantly reduce the bandwidth.

2.2. Record Format

Whenever an event is encountered, if tracing is enabled, an eight-byte record is written to the trace buffer. If that event happens to be a LDPCTX instruction, an additional 8-byte record is written to the buffer. This instruction occurs on a context switch to load process specific registers from memory into the processor. The basic eight-byte record consists of a 16-bit microsecond counter, a 16-bit instruction counter, the high 16 bits of the PSL, and information identifying the event. The extended eight-byte record contains the process and user IDs of the process being loaded as well as information identifying the file which contains the program that the process is executing (see figure 1).

The 16-bit instruction count field is the least significant word of a 32-bit internal instruction counter. This register is implemented through an unused internal register which is incremented by the microcode at the end of every instruction (unfortunately, the hardware does not provide such a register). We are relying on an event occurring at least every 65535 instructions. Since timer interrupts occur every 10ms (on both UNIX and VMS), the system would have to execute at 6.55 MIPS in order for this to be a problem. This translates to 1.91 cycles per instruction on an 8600 and 2.77 cycles per instruction on an 8650. While 65535 NOP's could probably execute this fast, the average number of cycles per instruction is 6 to 7, so this isn't much of a problem. Note, however, that if a more selective tracing scheme were implemented (i.e. in which not all events get traced), it might be necessary to store more of the 32-bit instruction counter.

The microsecond count field is the least significant word of a 32-bit microsecond counter. It gets incremented by hardware every microsecond and upon reaching -1 generates a timer interrupt and gets reloaded with -10,000. Thus, the least significant word is the only one that ever changes. The most significant word of the PSL really only contains three pieces of useful information: the IS bit (indicating if the processor is currently servicing an interrupt), the current and previous modes (kernel, executive, supervisor or user) and the IPL (interrupt priority level). UNIX only uses kernel and user modes while VMS uses all four. Software interrupts take place at interrupt priority levels 0 to 15 and hardware interrupts take place at levels 16 – 31. The SIRR is a register used to request software interrupts and the SISR is a register which indicates current pending software interrupts.

2.3. New instructions

The manipulation of the trace generation process and the retrieval of data are both controlled by new macro instructions. There are six new instructions in all which execute new microcode to exercise the data collection tools. These new instructions are defined to be opcodes that are unused in the VAX architecture. To use one of these instructions, a simple assembly language program can be written which executes the instruction, possibly moving registers between global variables. This assembly language routine can then be linked with a program written in a high level language. Alternatively, the new opcodes may be inserted at strategic places in assembly language routines. Figure 2 provides the complete semantics of the new instructions.

3. The Implementation of SPAM

A description of the 8600 microarchitecture and details of our microcode based measurement gathering environment is contained in [1]. In this section we will only describe the SPAM-specific parts of the implementation. Figure 3 provides an overview of SPAM microcode flows. The cycle previous to every macro instruction has the microorder "IRD". It is in this cycle that the microcode-maintained instruction counter is incremented. (Note that there are numerous places in the microcode which contain this microorder.) If no interrupt is pending, the next microcycle will be the instruction dependent entry point, otherwise, a microtrap will be taken to interrupt microcode.

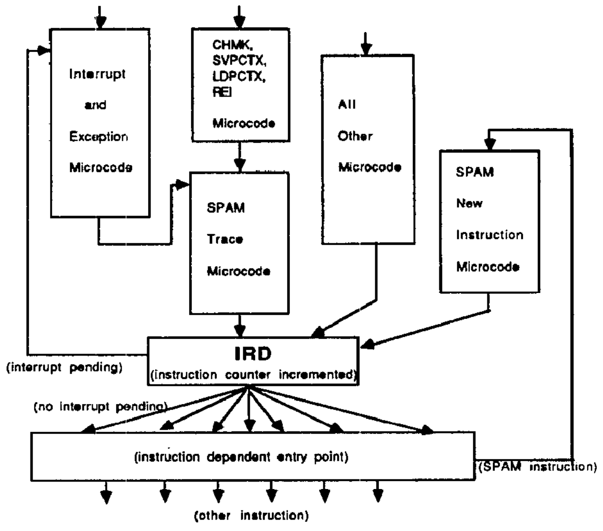
The SPAM trace microcode is logically inserted at the end of all traced flows. Where the instruction otherwise would have gone to an IRD state, it branches to trace microcode to create a trace record (if tracing is enabled). Note that if we want the instruction counter to accurately reflect the number of *entry points* rather than the number of *IRD* states, we must decrement the traced value on hardware interrupts. This can easily be done by the post-processor.

The trace buffer is a section of physical memory which is disabled before the operating system is booted, and then re-enabled by microcode afterward. Thus, the new SPAM instructions are the only way to read from the trace buffer. As far as the operating system is concerned, that memory doesn't exist. Note, however, that the cache could be significantly affected (especially since the 8600 has a write-back cache). We solve this problem by "translating" memory references. The microcode which reads and writes to the trace buffer remaps its addresses so that consecutive references are to the same set in the cache. This technique minimizes the effect on the cache of the tracing microcode.

Figure 2
New Instructions

Instruction	Opcode	Input	Output
GETSTAT	FD00	(none)	R11 = instruction count R10 = tracing status 0: tracing disabled -1: tracing enabled R9 = buffer read pointer R8 = buffer write pointer R7 = number of records in buffer R6 = buffer size in records R5 = buffer start address R4 = buffer end address
PAMMINIT	FD01	R11 = megabyte number R10 = slot number	(none)
ENABLE	FD02	(none)	(none)
DISABLE	FD03	(none)	(none)
READ_BUFFER	FD04	(none)	R11 = first longword of record R10 = second longword of record R9 = read status 0: OK, R10 and R11 valid -1: no data available R8 = tracing status 0: tracing disabled -1: tracing enabled
SET_PTRS	FD05	R11 = start address of buffer R10 = end address of buffer + 1	(none)

Figure 3
SPAM Microcode Flows

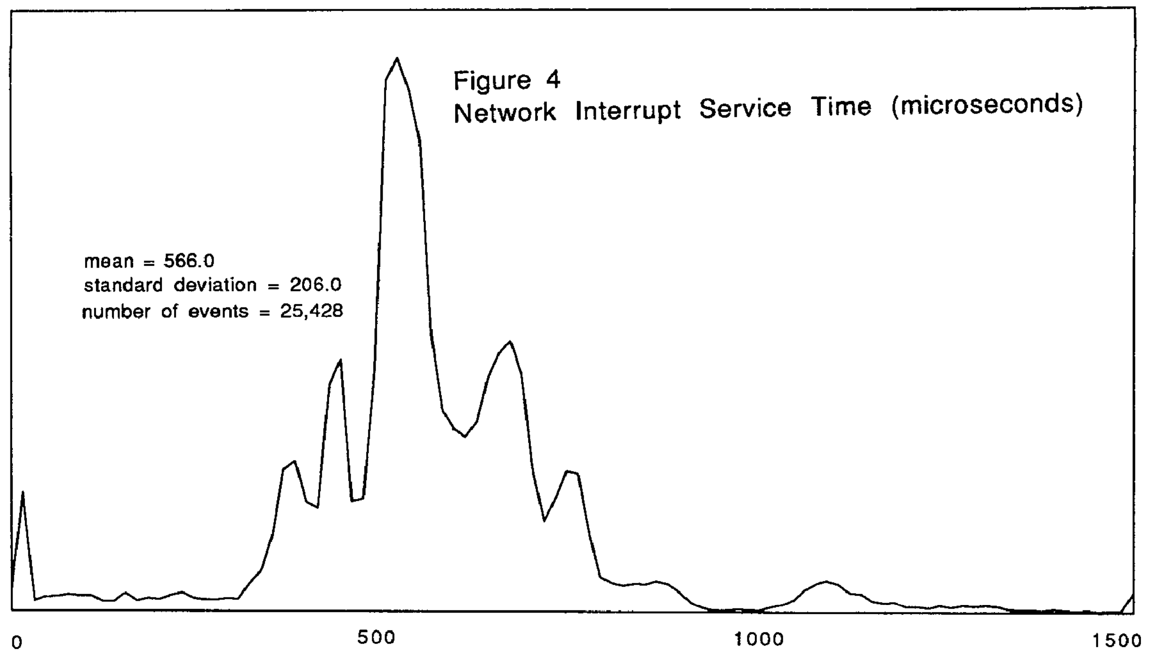

Interrupt and Exception Microcode
CHMK, SVPCTX, LDPCTX, REI Microcode
All Other Microcode
SPAM New Instruction Microcode
SPAM Trace Microcode
IRD
(instruction counter incremented)
(interrupt pending)
(no interrupt pending)
(instruction dependent entry point)
(SPAM instruction)
(other instruction)

4. Some Results

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 processes: the interrupt service routine and the network server process.

The CPU responds to a network event by generating a software level 12 interrupt. The interrupt handler then does the absolute minimal amount of work necessary to get the message from the hardware buffer to an operating system buffer. Then, a process (/etc/XNSrouted) 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 XNSrouted 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 which continuously reads SPAM records and generates a histogram of the time spent in the network interrupt routine. This program uses approximately 5% of the CPU, but it does not gen-



erate any network traffic, so the program itself doesn't affect the measurement. Figure 4 shows the results from a particular data collection run. It was taken over approximately a 4 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. In order to complete this analysis, we plan to set up SPAM to also measure the time spent by the XNSrouted process. We can simply write a program to recognize the loading of this process (based on inode) and accumulate statistics.

5. Conclusions

The use of microcode instrumentation to gather data for operating system analysis has many advantages. Unlike a purely software method, system perturbation is minimal and no messy hooks in the kernel are needed and unlike a hardware monitor, we have the flexibility, low cost and ease of use of a software system.

We have described SPAM in its current state, but the potential exists for many more features. In particular we are in the process of developing the user interface such that people involved in debugging, tuning and development of operating systems can manipulate SPAM to gather a wide variety of measurements without the need to be familiar with details of its implementation. We are making the trace record more general, and able to be specified by the user. The current record captures a core set of information that should be useful in most applications, but the need also exists to trace measurement specific data, for example a particular memory location. In addition, we are making trace triggering 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. Even this only scratches the surface, many more improvements and features are possible.

6. 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 help with the network interrupt example and 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] 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.