A Microcode-Based Environment for Non-Invasive Performance Analysis

Stephen W. Melvin
Yale N. Patt

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

ABSTRACT

We have developed an environment which allows us to collect data for performance analysis by modifying the microcode of a VAX 8600. This use of microprogramming permits data to be collected with minimal system perturbation (i.e. the data is almost as good as that obtained with a hardware monitor) but at the cost and with the ease of use of a software simulator. In this paper we describe the environment that we have developed and present two examples of its use. The first example, procedure call instrumentation, illustrates a technique for gathering data on how certain architectural features are used. The second example, instruction tracing, illustrates a technique for collecting data that can then be used in trace–driven simulation.

1. Introduction

One critical aspect of computer science is the evaluation of the performance of a computer system. Furthermore, as architectural mechanisms get more sophisticated, the data required to support this evaluation get more complex. For example, overall opcode frequency is sufficient to decide which instructions to optimize, but traces of addresses are needed to evaluate mechanisms such as caches and translation buffers. Even more comprehensive data is needed to evaluate more complex microarchitectural techniques, such as those that permit out–of–order execution. Thus, the need for sophisticated, high quality performance measurements is becoming increasingly more important.

An important issue for any type of measurement is the distortion to the data collected due to the act of taking the measurements. At one extreme is a hardware monitor, which can take measurements with virtually no effect on the system. However, hardware monitors are generally quite expensive. Also, physical access to the computer being measured is required as well as specific technical knowledge of the hardware configuration. At the other extreme are software simulators which can be easy to use and can be tailored to many different requirements. The problem here is that a software simulator typically handles only a subset of an entire system (e.g. a single process) and typically runs on the order of 1000 times slower than the hardware.

Most of the advantages of both hardware and software methods can be achieved with a measurement gathering technique that involves modifying the microcode of the system being measured. By so doing, measurements can be taken on a real computer system while it is executing jobs in real time. The amount of slowdown will vary depending on the type of measurement being taken, but in many circumstances it may not be noticeable. In cases where a large quantity of data is being gathered, the system may run 2–5 times slower, but this is still a lot better than a software method can achieve. In addition, microcode–based systems can be very flexible and easy to use. Once the core microcode is installed that gathers the information, everything else can be under software control.

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 [8]. In this two page note, Halbach discusses the gathering of trace information by using microcode modifications. Also, Armbruster discusses the gathering of instruction traces in [3], published in 1979. Grätsh and Kästner provide a brief history of firmware monitoring in [7]. Chroust, Kreuzer and Stadler discuss a microprogrammed page–fault monitor in [4] and Agarwal, Sites and Horowitz discuss a microcode–assisted address tracer in [1].

Our environment is based on microcode modifications to a VAX 8600. These modifications include additional machine level instructions as well as side effects to standard VAX instructions, but the VAX architecture is preserved. (Actually, there are a few exceptions where the VAX architecture is not quite preserved, as we will see later, but this is only to the extent that unused features are removed.) This means that all operating system functions and utilities can operate normally. Our environment runs under UNIX 4.3 BSD, but it is mostly operating system independent and many of the initial experiments were

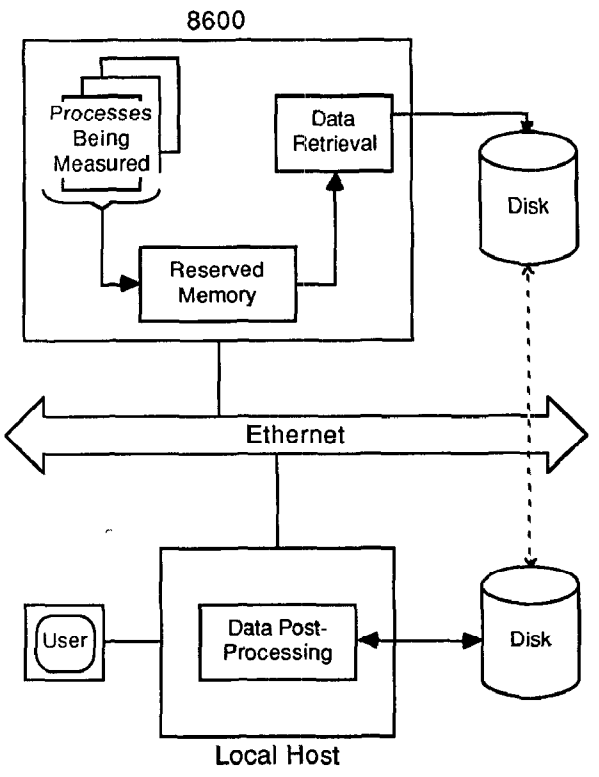

Figure 1.
User's View of the Environment

run under VMS. The user interacts with the environment by writing programs in C which are subsequently linked to special library files and are eventually loaded into ordinary binary files that contain the new instructions. Thus, no special privilege is required to compile or execute the program and the operating system doesn't require modification.

We use the term "environment" to describe our system because it is more than just a tool to gather a specific type of measurement. Rather, it is a set of microcode modifications along with software utilities that can be used to take a variety of measurements under the control of user-level processes. Thus, unprivileged users remotely logged in via the ethernet can exercise the environment to gather measurements for their own purposes, without disturbing any other system activity, except for the fact that only one user can be exercising the environment at a time. This is due to the fact that certain resources are used by the microcode which are common to the entire processor.

This paper is divided into six sections. Section 2 illustrates the environment as seen by the user. Section 3 provides an introduction to the 8600 microarchitecture and the modifications necessary to create the environment. Section 4 describes an experiment that was conducted to instrument VAX procedure calls. Some results are given for both VMS and UNIX. Section 5 describes a tool we are currently implementing that will gather instruction traces for a process, a set of processes or for the entire system. Finally, section 6 concludes with details of future work and some ideas about other projects that could be done.

2. User-Level Description

We will begin the description of our environment for performance analysis by looking at it from a user's point of view (see figure 1). The 8600 on which the environment is based is part of the Berkeley ethernet network. This means that users can remotely log in from other machines and can copy files back and forth between machines. From the user's point of view there are three separate activities taking place: the collection of data into a section of reserved memory, the transfer of the collected data from the reserved memory onto disk, and the post-processing of the collected data. The first two activities must take place on the 8600 while the third may take place on the 8600 or on the user's local host.

The collection of data and its retrieval from the reserved memory can both be controlled from the UNIX/C level. In both cases, programs are written in C which include calls to standard subroutines which exercise the environment. These programs are then loaded into binary files which are invoked from a UNIX shell. The set of processes for which the data is being gathered and the process which is collecting the data can be running concurrently. This would allow data to be taken continuously as long as the I/O system can keep up with the running process. If the rate of data accumulation is too high, the set of processes which are generating the data can be given low priorities in order to give them less of the processor and lower the rate of accumulation.

Measurements can be taken for a set of user processes (which would be the case if a user wanted to collect data about a specific program), or can include system activities over which the user has no direct control. In the latter case, in order for the user to control these system activities, special arrangements are necessary. If on the other hand, measurements are only being taken for processes that the user directly controls, other activity in the system is irrelevant. The one restriction that does exist in this case, however, is that only one set of measurements can be taken at a time because the microcode modifications manage resources that are common to the entire processor.

3. Modifications to the 8600 Microarchitecture

Figure 2 is a block diagram of the 8600 microarchitecture. It is partitioned into four major units that work concurrently, each performing a different part of the overall execution of an instruction. The IBOX prefetches the instruction stream, processes operand specifiers and passes operands and instruction-dependent control information to the EBOX, which is the main execution unit

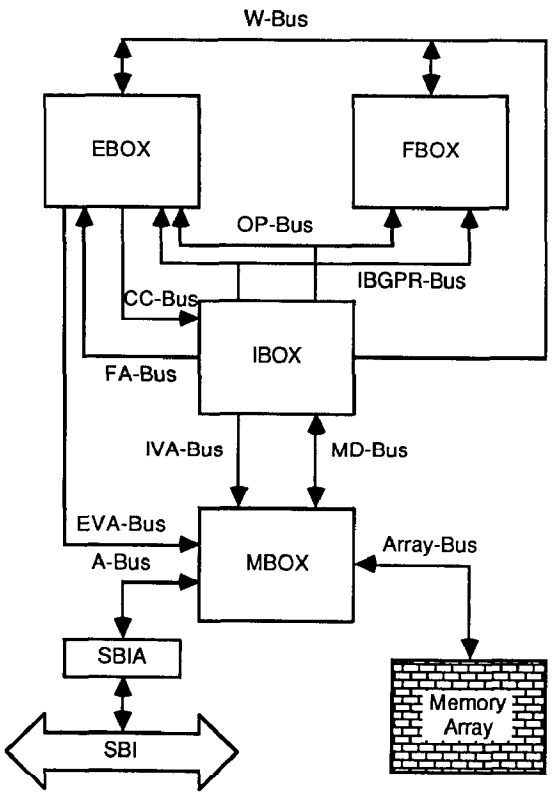

Figure 2.
8600 Block Diagram

of the machine. The EBOX is in charge of the overall execution of the machine and under exceptional conditions explicitly controls the other units. It also contains the main data path of the machine and most of the microcode. The MBOX processes memory requests from both the EBOX and the IBOX. It contains the translation buffer, the cache and the interface to the I/O subsystem. Finally, the FBOX contains special hardware to execute floating point instructions efficiently. The EBOX executes all VAX floating point instructions if the FBOX isn't present.

Figure 2 shows the main interface signals between the four major units. The MD-bus handles all data transfers between the MBOX and the rest of the system. There are three possible destinations for data flowing across the MD bus: the prefetch buffer, the IBOX port (used for operand processing), and the EBOX port (used for memory accesses during the execution phase of an instruction). The IVA and EVA busses provide the addresses for memory accesses to the MBOX. The op-bus is the main data path for operands from the IBOX to the EBOX and FBOX. The ibgpr-bus passes a GPR number to the EBOX, allowing it to access it's own copy. Thus, two operands can be passed to the EBOX in one cycle.

The w-bus can be sourced by the IBOX, the EBOX or the FBOX and is used mainly for the distribution of writes to general purpose registers (each of these three boxes has it's own copy of the GPRs). The w-bus is also used to transfer memory data from the EBOX to the IBOX. The fa-bus is a control bus that provides microcode entry points to the EBOX. Finally, the cc-bus provides the IBOX with condition code information computed in the EBOX which the IBOX needs for making branch decisions.

Since most of the microcode is in the EBOX and this is the only microcode currently involved in this project, we will examine the EBOX in more detail. Figure 3 is a block diagram of the EBOX data path. There are two main function units that can operate in parallel, a barrel shifter and an ALU. There is a 256 by 32 bit dual-ported register file (RA and RB) which can provide two sources to the function units and can receive a result in the same cycle. Of these 256 registers, roughly half are used to store constants while the other half are used to store the GPRs, internal processor registers and temporaries. The VMQ register drives the EVA bus and is loaded from the ALU.

Now let's look at the EBOX registers in more detail since they are important for the purposes of taking measurements. This is because they can be used to implement very fast counters. Information can be collected about the system without having to go to memory and with only an extra microcycle or two, thus slowing the

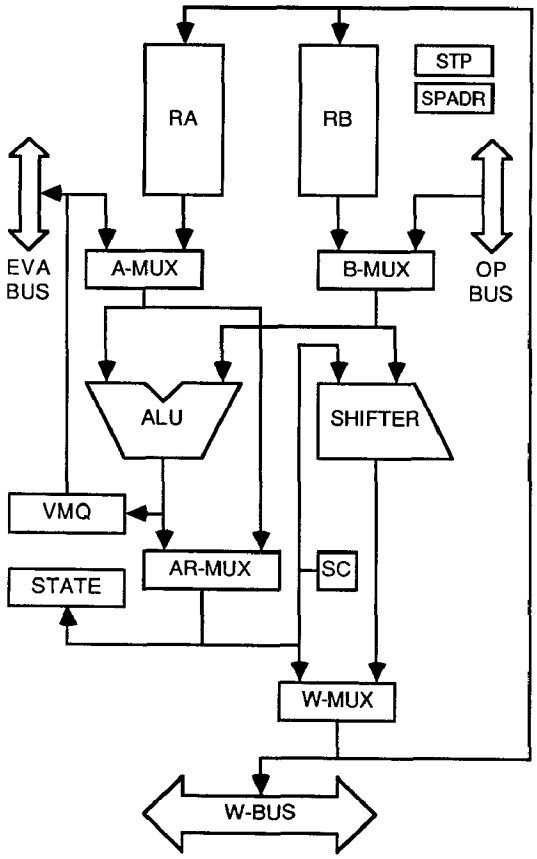

Figure 3.
EBOX Block Diagram

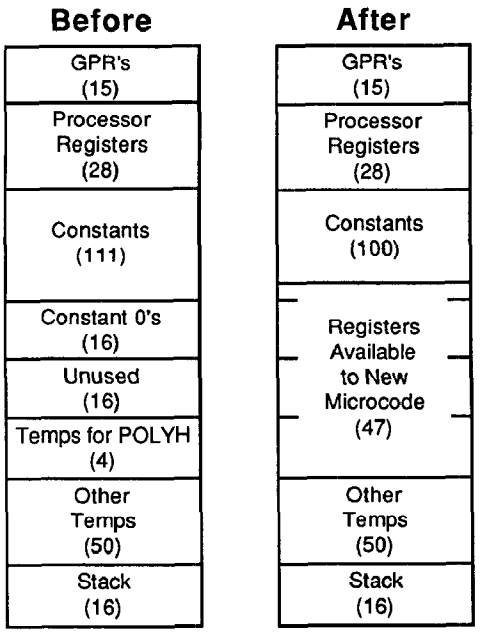

Figure 4.
Modifications to the EBOX Registers

processor down very little. Figure 4 shows the register map before and after the microcode modifications were made.

There were 16 unused registers in the initial system and we have been able to free up 31 more, leaving 47 registers available to the new microcode. One instance of making more registers available has to do with the POLYH instruction, which evaluates a polynomial on 128 bit floating point numbers. There are four temporary registers that are reserved for this specific instruction and unused by any other. Thus, if we are willing to do without POLYH, these four registers are available to the microcode.

Another way registers were made available was to eliminate a bank of 16 zeros. Because of certain register addressing restrictions, this bank sped up certain instructions by allowing a zero to be sourced when only the bank and not the register within the bank could be specified. By modifying the microcode in about 30 places (most of which required the addition of an extra cycle), and source the particular register that contains a zero, this bank was no longer needed. Finally, additional registers were freed by eliminating constants. Some constants are used in only a few places. By modifying the microcode to use other constants, adding a cycle or two where necessary, these registers become available.

Another important detail of the microarchitecture is the decode RAM in the IBOX. This is a memory, indexed by opcode, which specifies the access type and data type of each operand and the EBOX microcode entry points. The decode RAM, which is writable, has entries for 512 opcodes (all of the single byte opcodes plus all two byte opcodes that consist of hex FD followed by another byte). Since the VAX architecture only defines 306 of these opcodes, that leaves over 200 new instructions that can be defined with their own specific operand information and EBOX microcode entry points. This allows instructions that aid in the measurement gathering process to be defined and executed without disturbing the VAX instruction set. Thus, a particular process can be executing a secondary instruction set while at the same time other processes and the operating system are executing VAX instructions.

Finally, there is the issue of space for the additional microcode. The EBOX microcode source consists of approximately 75,000 lines of microcode in about 20 separate files which assembles to about 7700 microinstructions. Since the EBOX control store has space for 8192 microinstructions, this leaves about 500 for our use. We can actually get more space by removing parts of the microcode. For example, compatibility mode, which is implemented entirely in the EBOX and uses about 600 microinstructions can be removed. Our machine does not have to implement the entire VAX architecture, it just needs to execute the operating system and utilities necessary to support the environment. We have been running without compatibility mode and without POLYH without any problems.

The microcode source files are assembled and loaded into a binary format and placed on the console removable disk pack. These files are then loaded into the control stores before the operating system is booted. All control stores are in RAM, and any control store can be overlaid at the single microinstruction level. This allows us to assemble small files that patch the existing microcode in a fraction of the time it would require to reassemble the entire microcode. The control stores are not dynamically alterable. The CPU clock must be halted in order for them to be changed.

4. Procedure call instrumentation

One specific example of microcode-assisted performance analysis that we have performed is the instrumentation of VAX procedure call instructions. A lot of attention has centered on procedure calls and their implementation recently, in particular with regard to the RISC versus CISC paradigm. Furthermore, results can be greatly skewed depending on the benchmarks used. There are benchmarks that are virtually all procedure calls (e.g. Ackermann's function [13]) and benchmarks with practically no procedure calls. There are benchmarks that do a lot of register saving, and those that do very little. Thus, it is interesting to measure a real computer system, under different circumstances, to determine how procedure calls are used in practice. However, these measurements must be kept in perspective. It is dangerous to draw conclusions about architectural features in general from the results of a particular architecture, operating

total number of calls =	13980	
no pushes:	1119	8.00 %
1 push:	812	5.81 %
2 pushes:	580	4.15 %
3 pushes:	308	2.20 %
4 pushes:	5590	39.99 %
5 pushes:	832	5.95 %
6 pushes:	545	3.90 %
7 pushes:	413	2.95 %
8 pushes:	145	1.04 %
9 pushes:	840	6.01 %
10 pushes:	2796	20.00 %
11 pushes:	0	0.00 %
12 pushes:	0	0.00 %
R0 pushed:	0	0.00 %
R1 pushed:	0	0.00 %
R2 pushed:	12658	90.54 %
R3 pushed:	12049	86.19 %
R4 pushed:	11407	81.60 %
R5 pushed:	10927	78.16 %
R6 pushed:	5571	39.85 %
R7 pushed:	4900	35.05 %
R8 pushed:	4414	31.57 %
R9 pushed:	3766	26.94 %
R10 pushed:	2935	20.99 %
R11 pushed:	3630	25.97 %
alignment = 0:	13724	98.17 %
alignment = 1:	3	0.02 %
alignment = 2:	119	0.85 %
alignment = 3:	134	0.96 %
DV bit set:	0	0.00 %
IV bit set:	6	0.04 %
DV and IV set:	0	0.00 %
res. operands:	0	0.00 %

Figure 5.
Call Instrumentation Results
(1 hour of VMS idle time)

system, compiler, etc. They can, however, provide valuable information.

The two VAX procedure call instructions, CALLS and CALLG differ only in how the arguments are passed. Both instructions read the first two bytes of the target address to get the entry mask. This mask determines which registers to push onto the stack (after alignment) and how to set the DV and IV bits in the processor status longword. Control is then passed to the location following the entry mask. There are two bits in the entry mask which must be zero. If either bit is non-zero, a reserved operand exception occurs.

An exhaustive instrumentation of the CALLS and CALLG instructions would record every entry mask and the stack alignment before each instruction as well as the address of the call instruction and the address of the destination. However, since we wanted all data to be contained within the EBOX, a more restricted approach was necessary. We implemented 34 counters which are incremented based on the entry mask and the stack alignment.

First, there is a counter that counts the total number of call instructions encountered. Also, there are 12 counters which are incremented for each bit set in the register save mask. These counters count the total number of times a particular register is saved. In addition, there are 13 counters which are incremented depending on the total number of bits set in the register save mask, from 0 to 12. These counters provide us with a histogram of how many registers are saved. There are three counters that count DV and IV patterns of 01, 10 and 11 (the pattern of 00 can be obtained by combining these with the total counter). There are four counters that reflect the low two bits of the stack pointer, providing alignment information and there is a counter which counts the number of reserved operand exceptions.

In order to implement these counters, the microcode was modified in the CALLS and CALLG routines (most of which is shared). The counters for each bit in the register mask are incremented following the cycle in which the register is written to memory. Thus, one extra EBOX cycle per register pushed is required. The counters for the number of registers pushed are incremented by keeping a count of the number of bits set and then branching on it after all registers are pushed. This requires a total of three extra cycles. The stack alignment counters added one cycle as did the total counter, but the DV and IV counters were updated with no additional cycles. Also, there was an optimization for a mask of all zeros to allow the "no pushes" counter to be updated with only one cycle instead of three.

Thus, the overall effect is three extra cycles in the case of a zero mask, and five extra cycles plus one cycle for each bit set in the mask otherwise. The total number of cycles to execute a call instruction varies between about 10 and 25. Thus, each call instruction requires approximately 20% to 50% more EBOX cycles. Assuming calls represent 10% of all execution time, this translates to a slowdown of 2% to 5%. However, since call instructions are memory intensive, the actual slowdown is probably quite a bit less because the extra work is being performed while the EBOX would otherwise be stalled. (Note that since our counters are 32 bits wide, they can overflow after 4 billion call instructions; this takes many hours even on a heavily loaded 8600 and days or weeks on a lightly loaded one.)

The second part of the microcode modifications necessary has to do with retrieving the information in the counters. To accomplish this, we created new instructions that move the counters into GPRs. Since there are 34 counters, more than one instruction was necessary (four new instructions were defined). One possible problem is that the process can be interrupted or switched in the middle of the retrieval, in which case the values retrieved will be inconsistent. The solution is to move the total counter into a GPR on every instruction. Then, consistency can be checked by comparing the four totals retrieved. If they are not the same, the sequence is repeated.

Figures 5, 6 and 7 show the some of the results obtained from this experiment. Approximately one hour of idle time under VMS and UNIX are shown in figures 5 and 6 respectively. Figure 7 represents approximately 5 minutes of operation in which a large C program was being compiled by 30 separate processes running concurrently under UNIX.

5. Instruction Tracing and Program-level Analysis

Another performance analysis tool incorporated into our environment allows us to take very large instruction traces on a process by process basis. Traces are used extensively in many different areas. The most common are address traces, used for cache and translation buffer simulation. We are interested in instruction traces primarily in connection with our research in microarchitecture. These traces can be used to determine such things as the degree of local parallelism in an instruction stream and can be used to evaluate branch prediction algorithms.

An instruction trace consists of the dynamic instruction stream executed by the processor. If we assume that the code is not self-modifying and that the execution image is available, then the sequence of instruction addresses is sufficient. A post-processor can then retrieve the instruction stream. Also, we need to control which processes to trace, and when and how to turn tracing on and off. The VAX architecture has a convenient way to select processes to trace. Each process has associated with it a performance monitor enable bit (PME). It is unused by the operating system, but preserved in the process control block, so we can manipulate it and it will stay with the process across context switches and disk swaps. Thus, the microcode is written so that if the PME bit for the current process is set, tracing will occur, otherwise it will not.

total number of calls =	1754768	
no pushes:	271776	15.488 %
1 push:	144196	8.217 %
2 pushes:	585376	33.359 %
3 pushes:	89076	5.076 %
4 pushes:	74568	4.249 %
5 pushes:	531744	30.303 %
6 pushes:	24632	1.404 %
7 pushes:	0	0.000 %
8 pushes:	0	0.000 %
9 pushes:	0	0.000 %
10 pushes:	1	0.000 %
11 pushes:	0	0.000 %
12 pushes:	33436	1.905 %
R0 pushed:	34048	1.940 %
R1 pushed:	34048	1.940 %
R2 pushed:	34049	1.940 %
R3 pushed:	34049	1.940 %
R4 pushed:	34049	1.940 %
R5 pushed:	34049	1.940 %
R6 pushed:	64644	3.684 %
R7 pushed:	590232	33.636 %
R8 pushed:	663772	37.827 %
R9 pushed:	752848	42.903 %
R10 pushed:	1337156	76.201 %
R11 pushed:	1475196	84.068 %
alignment = 0:	1754768	100.000 %
alignment = 1:	0	0.000 %
alignment = 2:	0	0.000 %
alignment = 3:	0	0.000 %
DV and IV clr:	1754768	100.000 %
DV clr, IV set:	0	0.000 %
DV set, IV clr:	0	0.000 %
DV and IV set:	0	0.000 %
res. operands:	0	0.000 %

Figure 6.
Call Instrumentation Results
(1 hour of UNIX idle time)

total number of calls =	6433275	
no pushes:	1211828	18.839 %
1 push:	697949	10.849 %
2 pushes:	803285	12.486 %
3 pushes:	1576039	24.498 %
4 pushes:	766345	11.912 %
5 pushes:	605435	9.411 %
6 pushes:	746676	11.606 %
7 pushes:	0	0.000 %
8 pushes:	0	0.000 %
9 pushes:	0	0.000 %
10 pushes:	1	0.000 %
11 pushes:	0	0.000 %
12 pushes:	25717	0.400 %
R0 pushed:	25758	0.400 %
R1 pushed:	25758	0.400 %
R2 pushed:	25759	0.400 %
R3 pushed:	25759	0.400 %
R4 pushed:	25759	0.400 %
R5 pushed:	25759	0.400 %
R6 pushed:	811652	12.616 %
R7 pushed:	1377910	21.418 %
R8 pushed:	2144133	33.329 %
R9 pushed:	3720172	57.827 %
R10 pushed:	4523335	70.312 %
R11 pushed:	5182107	80.552 %
alignment = 0:	6433275	100.000 %
alignment = 1:	0	0.000 %
alignment = 2:	0	0.000 %
alignment = 3:	0	0.000 %
DV and IV clr:	6433275	100.000 %
DV clr, IV set:	0	0.000 %
DV set, IV clr:	0	0.000 %
DV and IV set:	0	0.000 %
res. operands:	0	0.000 %

Figure 7.
Call Instrumentation Results
(30 concurrent C compilations under UNIX)

In order to implement instruction tracing, several things are required. First of all, we need some reserved memory that the microcode can write into without causing problems for the operating system. The easiest way to do this is to reserve a section of main memory before the operating system boots [11]. The operating system will then configure itself without the reserved memory. In addition, we need to modify the microcode associated with the beginning of each instruction to write its address to the reserved memory. We also check for the end of the memory so that tracing can be disabled when the buffer is full. Finally, we created a set of new instructions that allow the reserved memory to be accessed and the tracing status to be controlled from a user-level process.

6. Conclusions

There are many different ways to collect data for performance analysis, each with its own advantages and disadvantages. We believe that the use of microcode assistance results in a system with significant advantages over other approaches and disadvantages which aren't as significant. Unlike a software simulator, a real computer system can be measured in real time running real problems and unlike a hardware monitor, we have the flexibility,

low cost and ease of use of a software system. However, it should be pointed out that there are limitations. There are certain things that are inaccessible to the microcode that could be measured by a hardware monitor. Also, the microcode environment is less flexible than a pure software one. It is implementation and microcode version dependent and it requires more time to implement and debug. Nevertheless, we feel that microcode–assisted performance analysis is useful in many circumstances and can provide valuable information to both computer architects and microarchitects.

There are many possible directions for this research. The establishment of an environment for a variety of performance measurements is currently our primary objective. The environment reported here allows measurements to be taken under a VAX/UNIX configuration. This is useful for many purposes but we are even more interested in other configurations. For example, how would performance be affected if only a subset of the VAX architecture were available? Or, how would performance be affected if the opcodes specified all of the operand addressing modes, as in many RISC architectures? This could be tested by writing a set of new instructions, giving them their own opcodes and executing programs that have been compiled to this new architecture. Certainly this is not the same as making a comparison between two different implementations that have been specifically designed for each architecture, but we feel that it would provide valuable insight.

7. 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. 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] W. Gregg Alexander, and David B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer*, Vol. 8, No. 11, November, 1975, pp. 41–46.

[3] 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.

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

[5] Douglas W. Clark, and Henry M. Levy, "Measurement and Analysis of Instruction Use in the VAX–11/780," *Proceedings of The 9th Annual Symposium on Computer Architecture*, April 26–29, 1982, pp. 9–17.

[6] Caxton C. Foster, Robert H. Gonter, and Edward M. Riseman, "Measures of Op–Code Utilization," *IEEE Transactions on Computers*, Vol. C-20, No. 5, May, 1971, pp. 582–584.

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

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

[9] Makoto Kobayashi, "Dynamic Profile of Instruction Sequences for the IBM system/360," *IEEE Transactions on Computers*, Vol. C-32, No. 9, September, 1983, pp. 859–861.

[10] Amund Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Communications of the ACM*, Vol. 20, No. 3, March 1977, pp 143–153.

[11] Richard L. Sites, personal communication.

[12] Reinhold P. Weicker, "DHRYSTONE: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, Vol. 27, No. 10, October, 1984, pp. 1013–1030.

[13] B. A. Wichmann, "Ackermann's Function: A Study in the Efficiency of Calling Procedures," *BIT*, Vol. 16, 1976, pp. 103–110.

[14] *VAX Architecture Handbook*, Digital Equipment Corporation, 1981.