Methodologies for Experimental Research in Computer Architecture and Performance Measurement.

Yale N. Patt

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

ABSTRACT

Since computer design is driven by the marketplace, rather than by some desire for internal elegance, the successful computer system often contains kludges that provide performance advantages over its competition. The result is that, while analytic treatments may prove helpful, to understand most computer systems requires the use of experimental techniques. In fact, as we have moved from preoccupation with cpu performance to an increased awareness of the behavior of the total system, the requirement for experimental work has become more pronounced. The purpose of this paper is to introduce the minitrack on experimental research. We discuss the various approaches to experimental research in computer architecture, reference examples of each approach, and identify a number of caveats. We conclude with an assertion about the need for the coordinated use of several experimental methods.

1. Introduction.

1.1. Goals.

Our goal is to implement computing machines that optimally use the available hardware and software technologies. If cost is a secondary consideration, this means the highest performance computers possible. If cost is a primary consideration, this means better cost effective computing engines. To succeed, we must first understand and then eliminate the system bottlenecks that prevent us from exploiting the available technologies.

1.2. Examples of what we want to know.

There are many potential causes of system bottlenecks. Among them, for example, are the inappropriate placement of the dynamic/static interface and the ill-tuned combination of the components of a complex system.

Computers are causal systems, yet the results we frequently hear quoted appear to be more magic than anything else. We [1] identified a critical design decision -- where to place the dynamic/static interface. We argue that there is an amount of complexity that must be dealt with in the transformation of high level language programs to the signals that control the gates and latches of the underlying data path. One can relegate that complexity to the compiler or to the hardware, but one has to deal with it somewhere. How does one make this choice? and perhaps equally important, can one implement a high performance system from uncomplex hardware and uncomplex compilers?

A modern computer system often consists of application software executing under the control of a distributed operating system executing on a loosely coupled connection of tightly coupled multiprocessors. How do we tune any of the components of such a system? If we partition the system into pieces, what system effects are lost by examining the piece outside the environment of the whole? With respect to total system performance, what is the advantage of one architecture over another, one operating system, one network topology, or one of any of the other components of the system over its alternatives?

1.3. Why experimental techniques?

Computer design is driven by market demands, whether it be time-to-market, price/performance, acceleration of some perceived important function, or software compatibility, rather than by some desire for internal elegance. Given the opportunity to retain internal elegance or to add a kludge that will provide an advantage over the competition, the successful computer systems designer usually opts for adding whatever it takes to sell more machines. The result is that, while analytic treatments may prove helpful, to understand most computer systems requires the use of experimental techniques.

In fact, as we have moved from preoccupation with cpu performance to an increased awareness of the behavior of the total system, the requirement for using experimental techniques has become more pronounced. While in some cases, the core of the cpu is simpler, total systems have become more complex. The cpu, memory system and i/o structure, the various components of the operating system, and the application software all influence total system performance.

Experimental work in computer architecture involves both simulated and real hardware executing real programs. The importance of simulation should not be minimized. It, too, is part of the fabric of experimental research in computer architecture, and has its place in developing an understanding of the bottlenecks to optimal performance. On the other hand, a whole lot of the problems attending a new computer architecture or implementation do not surface until one builds and performs experiments on real hardware. Moreover, even after a system is installed, the real bottlenecks to performance are clarified only when one performs careful measurements on that real system doing real work.

1.4. Organization of this paper.

This paper is organized in four sections. Section 2 discusses the various distinct approaches to experimental research, from software dominated trace-driven simulation to hardware intensive prototype build including hardware monitors. Examples of projects embodying each of the experimental methods are referenced. Advantages and disadvantages of each are identified. Section 3 describes a number of caveats attending this methodology. Section 4 offers some concluding remarks.

2. Types of experimental approaches.

2.1. Trace-driven simulation.

Trace-driven simulation allows us to collect data of whatever we wish without degrading the system being modeled, and to change easily the requirements for the data that we wish to collect. The method can be used with equal ease to collect data involving existing machines or experimental prototypes.

Trace-driven simulation has two major disadvantages which preclude its exclusive use as a measurer of system behavior -- it is slow and it is sanitized. The fact that it accumulates data several orders of magnitude slower than the system it is purporting to measure means one can not acquire in any reasonable way enough information to get a complete picture of what is going on. The fact that it is sanitized means that system effects which may not contribute to the data collected, usually influence the real system that is being simulated. Nevertheless, it is a good first step in gaining insights into system behavior. Furthermore, when coupled with other experimental methods which can validate the simulation model, it provides a mechanism that can give quick answers to direct the inquiry in one direction or another.

Examples of experimental inquiries using simulation are many, including the work in [2-4] that proved the HPS microarchitecture concept and the Motorola 88000 work included in these proceedings [5].

2.2. Software Monitoring.

Software monitoring, also, can involve either existing machines or experimental prototypes. Both can provide the same types of answers, although for prototypes, one must (obviously) first build the prototype. Software monitoring can be useful to validate a simulation model. That is, a comparison of data acquired by software monitoring with simulation results can lead to a better simulation methodology. Data can be acquired very rapidly relative to the time that is being measured.

Software monitoring is easy to do, and is often facilitated by architectural features such as a trace trap capability or a breakpoint instruction. In fact, of all the schemes, software monitoring is the most easily accessible of the experimental techniques. However, along with this ease of access comes the greatest challenge: to effectively make use of the data acquired by software monitoring is not so easy. The system effects are very much incorporated in the data, but not in a way that is easy to measure, in part because they are overwhelmed by the slowness of the software monitor doing the data collection.

Another disadvantage of software monitoring is the limitation, unlike simulation methods, of not being able to change the parameters of the system being measured. In some sense, one is stuck with the system one is running. For example, a two-way set associative cache can not be a direct mapped cache by changing a value as it can in trace-driven simulation.

Examples of software monitoring are contained in the work of [6], involving the monitoring of the Astronautics ZS-1, which are contained in these proceedings, and in the work of [7], involving the software monitoring of the Multiflow TRACE for measuring the potential value of decoupling the operations in a VLIW instruction.

2.3. Microcoded Instrumentation.

A half-way method between hardware monitoring and software monitoring is the technique of instrumentation using microcode. Like hardware monitoring, it is non-invasive, resulting in system effects not being clobbered by the monitoring device. On the other hand, like software monitoring, it is cheap and easy to change. Two recent examples of microcoded instrumentation are the operating system monitoring of the VAX 8600 [8] and the address traces obtained for the VAX 8200 [9].

2.4. Hardware Monitoring.

Hardware monitoring, like software monitoring, can involve either existing machines or experimental prototypes. In both cases the task is expensive, in dollars and in time. The clear advantage is that the information you obtain is pure, i.e., totally non-invasive. The disadvantage is that it is very expensive to build the apparatus in the first place, and equally expensive to change the measurements being acquired.

The classic work of Emer and Clark, which used a hardware monitor to acquire a day's information of the processing being done on a VAX 11/780, is an example of hardware monitoring [10]. Shebanow's instrumentation of the NCR four processor

Tower to collect address traces for parallel algorithms is another example of this method [11].

3. Caveats.

The strong encouragement toward the use of experimental data presented in this paper notwithstanding, one must caution against the misuse of data so collected. Rather than treat each concern in detail, I will reference four papers that consider the misconceptions that can result if experimental data is used without discrimination.

Levy and Clark [12] identified seven bad ways to compare computer designs, including: use small benchmarks that fit entirely within the cache, use limited data types, no i/o, and give unnecessary weight to the constructs that are present in the benchmarks, compare apples and oranges (i.e., compare simulations of one computer with software monitoring of another computer), use a particularly bad (or good) compiler. Colwell et. al. [13] examined the influence of various types of local storage (i.e., register windows, multiple register sets, standard register set) as compared to the influence of the instruction set on claimed performance for several different instruction set architectures. Smith [14] discussed the hazards of selecting the incorrect mean for a given set of data. Hwu [15] cautions the experimentalist on improper benchmarking. In addition we should add caution with respect to reaching generalizations from abnormal instances, and isolating the causal agent in a situation where several agents are active.

4. Concluding Remarks.

The purpose of this paper has been to focus on the need for experimental research in computer architecture and performance measurement, to discuss the various ways that this work can take form, and to identify the problems that can attend the careless use of experimental results. There is no question that the nature of our discipline makes the use of experimental work imperative, both before a machine is produced in order to get it right, and after the machine has been produced, in order to really get it right the second time around.

Our systems are complex. We can come up with a design, prove the fundamental concept it embodies by extensive simulation, and then build and test a system prototype, thereby verifying the concept and obtaining information useful for the next iteration of that design. The usefulness of one experimental method (simulation) is enhanced insofar as it reinforces inferences made using another method (program execution on a prototype).

I am reminded of the joke that ends with the punch line, "because the light is better over here." The reference is to the drunk who drops his key in the dark bushes next to his front door late at night, and is found searching for that key at the street corner four houses away, under the streetlight. The drunk would be better off back at the bushes next to his front door, perhaps with a piece of metal and the fingers of an ungloved hand. The piece of metal he could grope with, listening for the sound of a potential key. His fingers he could then use to feel the object found in order to determine whether it indeed was his house key.

The point, relative to the subject at hand, is that although analytic modeling may be useful as a first cut at understanding what is going on, the resulting data is far too sanitized to be of use for any but the grossest implications. Things actually happen in very intricate mutually interdependent ways, and to get a handle on what is going on, we need to do experimental work. In fact, we need to do this work at several levels of experimentation, taking advantage of the capabilities of each level, while understanding its limitations, and finally using the several levels to demonstrate the consistency of the results obtained. With this approach, we submit, we have a chance at understanding the bottlenecks of complex computer systems that prevent our optimal use of available technology.

References.

[1] Stephen Melvin and Yale Patt, A Clarification of the Dynamic/Static Interface, *Proceedings, 20th Annual Hawaii International Conference on Systems Sciences*, Kona, Hawaii, January 1987.

[2] Yale Patt, Wen-mei Hwu and Michael C. Shebanow, HPS, A New Microarchitecture: Rationale and Introduction, *Proceedings of the 18th Microprogramming Workshop*, Asilomar, CA, December 1985

[3] Wen-mei W. Hwu, Exploiting Concurrency to Achieve High Performance in a Single-chip Microarchitecture, Ph.D. Dissertation, Computer Science Division Report, No. UCB/CSD 88/398, University of California, Berkeley, January 1988.

[4] Stephen W. Melvin, unpublished report (PhD dissertation, University of California, Berkeley, expected May 1990).

[5] Trevor N. Mudge, Performance Studies of the MC88000, *Proceedings of the 23rd Hawaii International Conference on Systems Sciences*, Kona, HI, January, 1990.

[6] William Mangione-Smith, Santosh G. Abraham and Edward S. Davidson, The Effects of Memory Latency and Fine-Grain Parallelism on Astronautics ZS-1 Performance, *Proceedings of the 23rd Hawaii International Conference on Systems Sciences*, Kona, HI, January, 1990.

[7] EECS 570 Project Reports, unpublished, University of Michigan, 1989.

[8] Stephen W. Melvin and Yale N. Patt, The Use of Microcode Instrumentation for Development, Debugging, and Tuning of Operating System Kernels, *Proceedings of the 1988 ACM SIGMETRICS Conference*, Santa Fe, NM, May, 1988.

[9] Anant Agarwal, Richard L. Sites, and Mark Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, *13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, June, 1986.

[10] Joel Emer and Douglas Clark, A Characterization of Processor Performance in the VAX-11/780, *Proceedings of the 11th Annual Symposium on Computer Architecture,* Ann Arbor, Michigan, June, 1984.

[11] Michael C. Shebanow, unpublished report (PhD dissertation, University of California, Berkeley, expected May 1990).

[12] Henry M. Levy and Douglas W. Clark, On the Use of Benchmarks for Measuring System Performance, *Computer Architecture News,* December, 1982.

[13] Robert P. Colwell et. al., Instruction Sets and Beyond: Computers, Complexity, and Controversy, *Computer,* September, 1985.

[14] James E. Smith, Characterizing Computer Performance with a Single Number, *Communications of the ACM,* October, 1988.

[15] Thomas M. Conte and Wen-mei W. Hwu, Benchmark Characterization for Experimental System Evaluation, *Proceedings, 23rd Annual Hawaii International Conference on System Sciences,* Kona, HI, January, 1990.