CHAPTER 2

Methods

Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities 79
G. M. Amdahl

Evaluating Associativity in CPU Caches 82
M. D. Hill and A. J. Smith

A Characterization of Processor Performance in the VAX-11/780 101
J. S. Emer and D. W. Clark

2.1 Introduction

This chapter is not about computer architecture, per se, but about the methods used to evaluate alternative architectures. We include this introduction to methods because disciplined methods are essential for the advancement of computer architecture, as they are for all branches of science and engineering.

The chapter begins with a review of the scientific method and calls for its increased use in practice. It then discusses the three major classes of methods that computer architects use: analytic modeling, simulation, and system monitoring. Each section includes case studies to illustrate concepts. The chapter concludes with a discussion comparing methods and introductions to the included papers.

2.2 The Scientific Method

One model of innovation is that a person should come up with an idea (inspiration) and then argue orally or in writing for the idea. This approach works well in some fields, for example, philosophy and literary criticism. However, it works less well in science and engineering. In science and engineering, more progress has been made by subjecting new ideas to reality to see if they are better than existing ideas.

Francis Bacon crystallized this approach as the two-part *scientific method* in *Novum Organum* (1620). In the first part, the scientist comes up with a wild new idea with a flash of inspiration after years of delving into a problem. This idea is called a *hypothesis*. In the second part, the scientist puts the new idea to experimental tests to see whether it is actually better than existing alternatives. A hypothesis that has withstood the scrutiny of many experiments is called a *theory*. It is this second part that separates the scientific method from other methods of inquiry.

Applying the scientific method to engineering problems, such as in computer architecture, involves an additional degree of freedom. Principally, scientists seek to discover the phenomenon that is "out there," while engineers design new phenomenon. Technologies change, workloads change, and architectural ideas change. Thus, we can never do experiments to identify the best cache ever.

Nevertheless, architects are often lax in applying the scientific method. All too often, people develop an idea and then write some simulations to support the idea in one or more papers.

A more proactive application of the scientific method was presented by Platt [14]. Consider Platt's method applied to the problem of determining why a multiprocessor program runs so slowly:

- *Develop alternative hypotheses.* Hypothesis 1: The program has synchronization bottlenecks. Hypothesis 2: The program is taking too many cache misses. Having multiple hypotheses gives parallelism to the following steps and helps keep us from getting too attached to one hypothesis.
- *Develop one or more experiments that can exclude or corroborate an alternative hypothesis.* Experiment 1: Add code in every critical section that stalls for time T and counts how often it is executed. Can you develop experiments for Hypothesis 2?
- *Predict experimental results before running the experiment.* If Hypothesis 1 is true, a P-processor program that executes S stalls should slow by much

more than $T \times S/P$. If not, Hypothesis 1 is excluded.

- *Run experiments.* If the program runs only $T \times S/P$ slower, then Hypothesis 1 is excluded. We can continue with experiments for our alternatives or return to the first step and develop new hypotheses. If the program does run much more slowly, then Hypothesis 1 is corroborated. We should develop refined hypotheses (e.g., the synchronization bottleneck is for data structure A or B) and return to the first step.

Most computer architects agree in principle that using the scientific method is a good idea. Nevertheless, the literature is replete with examples in which authors see an effect, speculate about its cause, and move on without conducting an experiment to corroborate their speculation.

2.3 Analytic Modeling

Computer architects study computer systems with three basic methods: analytic models, simulation, and system monitoring. We now discuss each in turn.

Analytic models are an important—and currently underutilized—tool for understanding computer systems (and avoiding being drowned in data). Analytical models are mathematical expressions that approximate some behaviors of a system by capturing some system features and omitting others [10–12]. An accurate model predicts system behavior close to actual behavior. An insightful model omits irrelevant system aspects so that what remains captures the essence of what is important. Ideally, one prefers accurate, insightful models, but it is often worth trading some accuracy for much greater simplicity.

The rest of this subsection seeks to whet your appetite for analytic models by introducing some simple ones, hinting at more powerful techniques, and give three cache modeling case studies.

2.3.1 Three Simple but Useful Models

Analytic models vary from simple to complex. Three useful simple ones are Little's Law, simple queues, and Amdahl's Law.

Little's Law. This law applies to any stable (i.e., the number customers in the system does not go to infinity) system that can be modeled as a queue (i.e., customers arrive, wait for service, are serviced, and leave) [10, 12]. Little's Law says:

$$\text{Average number of customers in the system} = \text{Average rate customers leave} \times \text{Average time a customer spends in the system.}$$

Consider an application of Little's Law to a nonblocking cache. How many outstanding requests K should be supported to process a miss every 50 ns to a memory whose average latency (with contention) is 200 ns? The answer given by Little's Law is four requests = (one request/50 ns) × (200 ns). In many cases, however, one would want more than four to handle "bursts" of requests. Simple queues provide a way to model burstiness.

Simple queues. Consider a model of a queue to a single server, where the queue size does not grow to infinity (i.e., the queue is stable) and customers arrive at a time independent of the current time (stationary) as well as independent of the number of customers already enqueued (open) [10, 12]. Let *throughput* be the rate customers leave this queue and *utilization* the fraction of time the server is busy. Then, for this so-called G/G/1 queue:

$$\text{Throughput} = \frac{1}{\text{Average time between customer arrivals}}$$

$$\text{Utilization} = \frac{\text{Average time to service a customer}}{\text{Average time between customer arrivals.}}$$

Often, we can further assume that customers arrive at a time independent of past arrivals (Poisson or Markovian arrivals) and service times are distributed exponentially (Markovian service times).[1] These additional assumptions create an M/M/1 queue and allow us to estimate *average latency*—the time from when a customer arrives to when it leaves—with a simple equation:

$$\text{Average latency} = \frac{\text{Average time to service a customer}}{1-\text{Utilization.}}$$

To see the utility of an M/M/1 queue, consider designing a simple nonpipelined bus where requests will arrive "at random" about every 50 ns, and you are supposed to service them with a latency of no more than 55 ns. If you pretend that requests come in at precisely

[1] An exponential service time distribution means that the probability that a particular service time is less than x is $1 - e^{-x/s}$, where s is the average service time. In practice it is not critical that service times be distributed exactly exponentially. It is only important that many service times are not much larger than others. In particular, the following result is only slightly pessimistic approximation of a system where all service times are exactly equal (deterministic).

50-ns intervals, designing a bus with a service time of 49 ns seems sufficient. If you do this and requests come in at random, bursts will cause the actual latency to be about 2450 ns (49/[1 - 49/50]). To handle bursts, the M/M/1 queue predicts you should instead "over"-design the bus by a factor of two, because a service time of 25 ns yields an average latency of 50 ns (25/[1 - 25/50]).

An important corollary of M/M/1 queues is that one can design for high utilization or low latency but not both, as long as arrivals come at random. A queue with 90% utilization has a latency 10X worse than the average time to service a customer, whereas a queue whose latency is 10% worse than the average time to service a customer has a utilization of 9%. Improving utilization and latency together requires that arrivals not be random (e.g., by using schedules).

Amdahl's Law. Amdahl's Law is so well known that people sometimes forget how widely applicable it is [2]. Consider a system that originally spends F fraction of time, $0 \leq F \leq 1$, doing function X. Consider an enhancement that speeds up function X by:

$$S_X = \frac{\text{Time to do } X \text{ originally}}{\text{Time to do } X \text{ with enhancement}}$$

If the new system performs X at the same frequency, then Amdahl's Law predicts:

$$\text{Overall speedup} = \frac{1}{([1-F] + F / S_X).}$$

For example, let's calculate the overall speedups for (1) a factor of ten improvement in a function used 5% of the original time and (2) a 10% improvement of something that operates 95% of the time. Plugging values into Amdahl's Law reveals speedups of 1.047 for (1) and 1.094 for (2). Therefore, attack the common case first. Furthermore, by taking the limit as S_X goes to infinity, we get:

$$\text{Overall speedup} \leq \frac{1}{1 - F}$$

Thus, the improvement because of (a) is bounded by 1.052.

2.3.2 More Powerful Modeling Techniques

The academic literature is replete with examples of more powerful modeling techniques. These techniques can model effects that are very subtle but that often require more mathematical sophistication to develop and use. If you are a student, you may benefit from taking a few courses in probability, statistics, renewal theory, Markov chains, and queuing theory.

We do not have the space here to introduce complex modeling techniques at a level of detail that would allow the reader to use them. Instead, we will whet your appetite for two modeling techniques, Markov chains and queueing networks, and one solution technique, customized mean value analysis.

A Markov chain describes a system with n states where the probability that the next state is j given that the current state is i is p_{ij} regardless of all previous states [10–12]. Markov chains permit simple analysis: for example, to determine the steady-state probability of being in state i. Consider a cache coherence protocol (with states invalid, shared, and exclusive) where we have measured the state transition frequencies. If we further assume that one spends about the same amount of time in each state, a Markov chain can be used to determine the steady-state probability of being in each state.

Queuing networks are a very powerful model technique [10–12]. They are constructed by interconnecting service centers. Each service center has one or more servers (one, k, infinite) that serve customers with some time distribution (exponential, deterministic) after some queuing discipline (first-come-first-serve, last-come-first-serve). A center with one server using exponential service time after a first-come-first-serve queue, for example, would be like the server in an M/M/1 queue. Unlike an M/M/1 queue, however, arrivals are not likely to be Markovian but are instead determined by the rest of the queuing network.[2] One can use a queuing network to model a symmetric multiprocessor memory system as follows: Have a service center for each processor, each cache, each snoop, the bus, and the memory. Have cache misses leave the cache, arbitrate for the bus, use snoops and memory, and so forth. Probabilities can be assigned based on the type of bus request and where it finds the data.

Once one has a queuing network, the next step is to "solve" it to answer questions like, "What is the average memory latency?" This can be done analytically for a restricted class of networks called product-form or separable networks. Alternatively, one can always simulate the network. In many cases in computer

[2]There are, however, a useful class of queuing networks—product form networks—whose structure lets one treat arrivals as Markovian to greatly simplify solving for many important properties.

architecture, customized mean value analysis provides the most attractive solution method.

Customized mean value analysis (CMVA) works for a large class of queuing networks but can only provide mean values [12]. Thus, it can yield the average memory latency, but not the distribution of memory latencies. The basic idea is a write a set of custom equations that describe what a customer must endure. A cache miss, for example, must go to the bus and may go to other processors and memory. At each center, the customer endures some mean queuing delay and some mean service latency. Mean service latency is easy to calculate. Mean queuing delay is calculated by assuming customers arrive "at random" when they are able to arrive. Sometimes these equations can be solved in closed form, even though most commonly simple iterative methods suffice. CMVA has been successfully applied to many systems. For a recent example, please see Sorin et al. [20].

2.3.3 A Case Study: Three Cache Models

We next present three cache models to illustrate how modeling can be productively employed at various levels of detail to provide varying levels of insight and accuracy.

Hill's 3C model of cache behavior is an example of a very simple model [6]. The 3C model partitions cache misses into *compulsory misses* (that must occur in any cache), *capacity misses* (because of a cache's finite size), and *conflict misses* (from restricted associativity). The model defines miss type operationally. The conflict miss ratio, for example, is the miss ratio of a cache less the miss ratio of a cache of the same size and block size that is fully associative and uses least recently used (LRU) replacement.

The 3C model is simple. It provides some insight even without numbers. Jouppi, for example, credits it for helping him come up with victim caches and stream buffers [8]. On the other hand, the 3C model does not provide much accuracy or predictive power. Measuring the 3Cs for a direct-mapped cache, for example, will not let you predict the miss ratio for a two-way set-associative cache.

Smith's set-associative model is less simple but has more predictive power than the 3C model [6,19]. This model asks:

> Why do set-associative cache perform worse than fully associative ones? Is it that real workloads have pathological conflicts? Or is it just that if you uniformly distribute references among C sets of small size A, by random chance some sets will get too many references?

Smith's model examines these hypotheses. He uses Bayes' rule to translate fully-associative miss ratios for caches of every size into set-associative miss ratios, assuming misses map independently and uniformly to all sets. Fully associative simulation data is fed into the model so that it can make set-associative predictions. These predictions are then compared with set-associative simulation data. The results are an excellent fit. Thus, the experiment corroborates the hypothesis that "random" behavior is sufficient to explain the difference between set-associative and fully associate miss ratios.

Smith's model has predictive power and can be accurate, but it is limited to matters of associativity. It would be nice to have a model that was comprehensive. Arguably the most successful comprehensive cache model is by Agarwal, Horowitz, and Hennessy [1]. This model has components for start-up effects, non-stationary behavior, intraprogram interactions, and interprogram interactions. Within these, intuitive equations model such things as set-conflicts and spatial locality. Parameters for the model are gathered from simulation data. Then the model is validated against other simulated configurations. The model achieves good accuracy and excellent relative accuracy across variations in cache size, associativity, block size, and multiprogramming degree.

In our opinion, these results make this model the most successful comprehensive cache model ever. Why then isn't every cache designer required to learn it? The problem is that this model is too complex to provide insight to a large audience, and it has too many inputs to be practical for specific predictions. In particular, gathering all the input parameter values requires a simulation infrastructure capable of gathering all the results. A lesson for model designers is that one should consider both insight and accuracy.

2.4 Simulation

Simulation is arguably the preferred methodological tool of computer architects. With *simulation*, a computer program—called the simulator—running on a *host* computer is used to mimic the functionality, and usually some performance metrics, of a *target* computer system.

2.4.1 Alternative Levels of Abstraction

Simulations can be performed at many levels of abstraction. At a very high level, one can simulate an analytic model to solve it. This level blurs the distinction between analytic modeling and simulation and, perhaps, is best considered a solution technique for analytic models.

Functional. A *functional* (or *architectural*) *simulator* mimics at least the application binary interface (ABI) of the target computer system. Complete functional simulators will also mimic how the hardware appears to the target operating system, whereas less-complete ones will only approximate a subset of the ABI. In any case, the goal is to be able to run target software correctly as it reads target inputs and produces target outputs. Functional simulators also often produce workload metrics, such as number or instructions or floating-point multiplies executed. Functional simulators operate on a workload consisting of one or more target programs with associated target inputs. This type of simulation is called *execution-driven*, because the workload executes during simulation and can be affected by simulation. In a multiprocessor simulation, for example, changing the cache size can affect which thread wins the race to a lock.

A simulator's *slowdown* is the time is takes to execute a simulation of some target software on a host of comparable power to the target divided by the time it would take to run the target software on the target. Smaller slowdowns are good and a slowdown of 1 is ideal. Functional simulators have slowdowns between 1 and 10.

Microarchitectural. The next level down in simulation is *microarchitectural simulation*, the bread-and-butter technique of computer architects. A microarchitectural simulation mimics the behavior of microarchitectural features, such as caches, memory banks, branch prediction tables, pipeline bypass stalls, and reservation stations. It produces performance metrics, such as cycles to execute a program and cache miss ratio. Microarchitectural simulations can be execution driven (i.e., present an ABI to target software) or trace driven. A trace is a log of relevant workload activity (e.g., a series of dynamic memory references). Trace generation is performed prior to simulation, for example: by system monitoring. *Trace-driven simulation* consumes the trace as it executes. Traces allow trace generation and simulation to be decoupled but preclude the simulation from affecting the traced workload.

Microarchitectural simulations may model aspects of computer system microarchitecturally, functionally, or not at all. An execution-driven simulation concentrating on cache behavior, for example, could accurately model cache microarchitecture but just mimic the function of the rest of the processor. A trace-driven cache simulator might omit the rest of the processor altogether. Depending on the level of detail, microarchitectural simulators suffer slowdowns of 10–10,000.

Arguably, the most powerful current microarchitectural simulator is Stanford's SimOS [17]. SimOS can simulate SGI Challenge-like multiprocessors with functional fidelity sufficient to boot a commercial operating system and run a commercial database. SimOS supports alternative modules that enable different aspects of a system to be modeled at different levels of detail at different times. One can, for example, use fast functional simulation to boot the operating system and initialize applications.

Gate-level. A gate-level simulator is one that is sufficiently detailed to model the actual structure of hardware. Is a barrel shifter implemented with multiplexors or pass transistors? How large are control program logic arrays? Gate-level simulation of a complete computer system for even a few cycles is very hard, and operation on a sensible workload is usually impractical. Instead, architects and implementors use gate level simulation for a few aspects of a system while modeling other aspects functionally or not at all. When many aspects of a system are not modeled, a challenge is providing workload inputs to a gate-level simulation. This problem is often called *test vector generation*.

Circuit simulation and below. Even lower down are analog circuit simulations, such as SPICE. These simulators are typically applied to isolated aspects of computer systems, such as adder carry chains, register cells, and long-line drivers. At an even lower level, one can simulate based on device physics, which is becoming increasingly necessary for dynamic memory design.

Summary. In practice, many simulations combine components from different levels. High-level components are used when simulation speed is important, detailed aspects have not yet been designed, or this aspect of the design is not the current focus of activity. Low-level components are used otherwise. Multiple level of the same component can also be run in parallel to provide evidence that the lower level implementation matches the higher level specification.

2.4.2 Case Study: Microarchitectural Simulation of Memory Hierarchies

Memory systems have long been the focus of microarchitectural simulations, because those simulations have

been so effective at improving memory hierarchies. We can learn about simulation challenges by considering how memory system simulation has evolved from the first cache paper by Wilkes [22] that had no numbers to recent multiprogramming multiprocessor memory system evaluations [16, 17].

Trace-driven memory system simulation was the method of choice through at least Smith's classic survey [18]. These simulations omit all aspects of a computer system but its cache. They model the workload as a trace that exists prior to simulation. A *trace* is a long sequence of tuples (e.g., a million or more) representing the dynamic memory references of a workload. Each tuple contained an address (virtual or physical), type (read, write, instruction fetch), and sometimes length (e.g., 4 bytes). Traces are large and stored on tape or disk. Trace-driven cache simulators model the state and tags of a cache and nothing else. In particular, the processor, cache data, or memory does not have to be modeled.

Nevertheless, trace-driven cache simulations are exceedingly time consuming, especially because one typically wants to evaluate the performance of many alternative caches. Mattson et al. [13] attack this problem in a seminal paper that finds ways to evaluate multiple alternative caches in one pass through an address trace. This work is summarized in this reader in Hill and Smith [6] in a way we think is more accessible than the original work. The basic idea can be illustrated by considering alternative fully associative caches that use LRU replacement (and have the same block size and do no prefetching). The state of all the alternative caches can be modeled with a single linked list, where the head points to most recently used (MRU) block and the i MRU block points to the $i + 1$ MRU block. For example (from Fig. 6 [6]):

$$6 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 0 \rightarrow 7 \rightarrow 2 \rightarrow 8.$$

If the next reference is to block 2, this technique—called *stack simulation*—finds the block at *distance* 7 and updates the list to make 2 most recently used:

$$2 \rightarrow 6 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 0 \rightarrow 7 \rightarrow 8.$$

Caches of size 7 and larger hit, whereas caches smaller than 7 miss.

Stack simulation can easily be extended to simultaneously simulate alternative caches with any fixed number of sets and to so-called *stack replacement algorithms*, which include random and Belady's optimal, but not first-come-first-serve. Hill and Smith extend stack simulation with *all-associativity simulation*. A single all-associativity simulation run can evaluate alternative caches of all sizes and associativities with LRU replacement, some restrictions on set-mapping functions, and a fixed block size. Thus, a designer can evaluate all caches about a design point with one simulation per block size of interest!

At this point, it might look like memory hierarchy simulation research could stop. But, as always, designs move forward and whittle away at the effectiveness of methods. First, single caches were replaced with two levels of caches. Fortunately, Przybylski [15] showed that caches whose size differs by a factor of eight or more could be simulated independently. This result is practical, because level-two caches are almost always at least 8X larger than level-one caches. Second, level-two caches became very large (e.g., $\geq$ 1 Mbytes) and could be evaluated only with very large traces (e.g., giga-references per program). This has led researchers to considered sampling in time and space (cache sets) to reduce trace size [9]. Third, caches began to employ more timing-dependent behavior, such as prefetching and nonblocking support. Evaluating the effectiveness of these features requires the modeling of timing both in the incoming workload and in the memory system. Fourth, caches are increasingly integrated with speculative out-of-order processing engines, further obscuring the relationship between miss ratio and program execution time. These trends are encouraging the elimination of traces in favor of execution-driven simulation that models both the cache and processor. The desire for results of higher and higher fidelity is forcing both the cache and processor to be modeled at a detailed microarchitectural level. Finally, increased interest in P-processor multiprocessors has multiplied simulation requirements by P and added tricky interaction cases [17].

In summary, new designs will require new methods. Understanding how past methods have evolved in present methods facilitates developing new methods from existing ones.

2.5 System Monitoring

George Santayana said, "Those who cannot remember the past are condemned to repeat it." System monitoring is the technique that computer architects use to learn from the present (soon-to-be past). With *system monitoring* (often called *hardware monitoring*), one records information about the behavior of a running system. Examples of system monitoring include using

an oscilloscope to record bus transactions, reading hardware performance counters, and modifying an executable to output information at procedure call sites.

Even though system monitoring can take many forms, its practice shares several unifying features. Most important is that the system to be monitored must exist. Thus, system monitoring can give us incredible detail about current hardware and software, but extrapolation to systems must be done with other methods. Nevertheless, most new systems use evolutionary hardware extensions to run evolutionarily new software. In these situations, system monitoring can provide valuable information on what to do. Equally importantly, system monitoring can tell us what not to do. An optimization may sound good, but system monitoring can reveal that the optimized situation does not occur often enough to be bothered with (remember Amdahl's Law).

2.5.1 Monitoring Mechanisms

System monitoring requires monitoring mechanisms. *Monitoring mechanisms* reveal system information to the monitor. The classic monitoring mechanism is the oscilloscope probe. In the days of medium-scale integration, such probes could access most of a system's architectural and microarchitectural state. Today, one can typically access only level-one cache misses (with great difficulty because of high frequencies) and system bus transactions. Tomorrow, probe access will likely be even more limited. Recognizing these limitations, hardware designers are now adding explicit monitoring mechanisms. Current examples include register- and memory-mapped cycle counters, cache miss counters, and bus-transactions-of-type-X counters. Future machines are likely to include an even richer array of monitoring mechanisms as performance optimization becomes more important and die area less precious. Finally, system monitoring mechanisms can be software only. On one hand, this can take the form of small changes to existing software to record activity (e.g., augmenting a software TLB handler to log TLB misses). On the other hand, it can mean a wholesale rewriting of executables to augment them with monitoring functions (e.g., as can be done with ATOM [21]).

2.5.2 Perturbation, Repeatability, Representativeness, and Sampling

Three important concerns of system monitoring are perturbation, repeatability, and representativeness. *Perturbation* occurs whenever the behavior of the system being monitored is not statically identical to an unmonitored system. A program-counter trace of an executable augmented by ATOM, for example, is not the same as the program-counter trace from an unmodified executable. On the other hand, reading a cycle-count register before and after a long-running application will cause negligible perturbation. In many cases, one can mitigate the effects of perturbation by compensating. The augmented executable, for example, can easily be modified to emit the program counters of the unmodified executable. Other cases are not as trivial. If executable editing slows applications by a factor of two, for example, then disk accesses and clock interrupts will appear twice as fast. One can compensate by delaying disk accesses and clock interrupts, but is this enough? The bottom line is that the careful system monitor hypothesizes that he or she is perturbing the system and tries to gather evidence to contradict this hypothesis.

System monitors must also be concerned with *repeatability*. Real systems may not run the same way twice because of different initial conditions or different concurrent activity. The number of conflict misses an application suffers in a physically-index level-two cache, for example, can be affected by what physical page frames are on the free list. There are two tools for dealing with repeatability. First, one can minimize potential causes of variation. Most commonly, this is done by rebooting the system and disallowing concurrent activity. Second, one can perform multiple runs of the "same" case. In some cases, one can informally verify that the variation between runs of the "same" case are much smaller than the variation between runs of different cases. In other cases, one should turn to the statistical technique developed for this issue: *analysis of variance* [7]. The careful system monitor hypothesizes that the effects he or she is measuring result from random variation and then tries to gather evidence that they are instead due to systematic effects.

Finally, the data gathered with system monitoring should be *representative* of the *population* of all possible data. Because the full data is typically exceedingly large, one is forced to collect some number of *observations* to form a *sample* of the population. The process, called *sampling*, can be implicit or explicit. When one selects twelve programs to be monitored, one effectively takes a sample from the population of relevant programs and implicitly assumes that these programs are representative of a much larger population of programs. Common sense must be applied here. Is SPEC95 representative of the population of programs my machine will run? If the

answer is "no," all the other wonderful work in a system monitoring experiment is worthless.

On the other hand, sampling is explicit when we repeatedly collect bursts of cache miss addresses. Statistics allows us to make very powerful inferences from unbiased random samples to the population in general (e.g., the mean of a sample tends to the mean of the population with some expected variation). An observation within a sample is unbiased if it accurately captures the system feature being measured. A snapshot of consecutive level-one cache miss addresses, for example, will not exactly yield the level-two cache miss ratio for the snapshot, because the level-two cache state at the beginning of the snapshot is unknown. This effect is mitigated with a snapshot using set sampling that takes the same number of cache misses from fewer sets. Whereas care must be taken to corroborate that the sampled sets are representative of all sets, set sampling has proven very effective for studying large caches [9].

2.5.3 Analysis

The final step of system monitoring is analysis of the data obtained from the monitoring mechanisms. This data is often low level (e.g., program counter virtual addresses) and voluminous. The goal of analysis is to transform the data into insight. Analysis should make data more high-level (e.g., by mapping program counters to procedure names). It should also reduce volume by transforming raw data into averages, histograms, or distributions. Care must be taken to mitigate the chance that analysis destroys evidence of effects. Procedure names may obscure an effect related to an address level-two cache set. Averaging can hide bimodal distributions. Once again, careful application of analysis requires that one assume he or she might be obscuring effects and that he or she gather evidence that this is not the case.

2.5.4 Case Study

For an excellent case study of system monitoring, please see Emer and Clark's "A characterization of processor performance in the VAX-11/780" [4]. This paper is included in this reader and introduced in section 2.7.3.

2.6 Discussion

Next we compare methods and touch upon selected areas not covered in this chapter.

2.6.1 A Comparison of Methods

Analytic modeling, simulation, and system monitoring are complementary tools for computer architects. Analytic models can provide insight long before a system exists. Simple models are useful by anyone. More-complex models require expertise to construct, but can be built and solved rapidly. Analytic models, however, cannot model all details of a system. Thus, other methods are needed to examine more-detailed effects and verify when analytic models are sufficiently accurate.

Simulation is the principal tool of computer architects. Simulations can model any system to any required level of detail. Simulators, however, require much more work to construct than do analytic models. In our view, computer architects currently put too much faith in simulation. Simulations are believed without much scrutiny as to whether the simulation code has bugs or just plain assumes away important effects. Whereas analytic models make assumptions explicit, simulator assumptions are often buried deep in proprietary code. We recommend that computer architects put less faith in simulators that have not survived the scrutiny of some verification.

Finally, system monitoring is the backward-looking method. It requires the system to exist, can rarely vary the system in many ways, and is usually hard to do. Nevertheless, it provides a wealth of data that is true for at least one real system and is useful to similar systems.

2.6.2 Important Aspects Not Included

Any short introduction and three papers on methods must omit important aspects. A few of these are discussed here.

Experimental design. Designing an experiment is harder than this chapter's discussion implies. For example, one must learn to control random error, make sure all important factors are considered, and isolate the effects and interactions of factors. We refer the reader to Jain [7, pt. IV].

Benchmark selection. As eluded to earlier, benchmark selection is critical to avoid "garbage-in-garbage-out." Good discussions of benchmark selection can be found in Hennessy and Patterson [5] and Culler, Singh, and Gupta [3].

Sizing and scaling. What do results mean if an application's input size is scaled down to permit simulation to complete in reasonable time? Can we scale down cache sizes to compensate? If this program were run on really fast systems, would users want the same answer faster or a more detailed answer in similar time? To answer these and related questions, we direct the reader to Culler, Singh, and Gupta [3].

Computer-aided design (CAD) and testing/verification. Finally, this chapter ignores the many issues that surround the design, verification, and testing of computer systems.

2.7 Discussion of Included Papers

We have selected three sample papers to illustrate models, simulation, and monitoring.

2.7.1 Amdahl's "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities" [2]

Amdahl's three-page paper presents what is now called "Amdahl's Law." Amdahl's Law appears in the prose and not as an explicit equation. It is part of an argument defending why uniprocessors will survive the current (in 1967) threat from multiprocessors. Amdahl was right—uniprocessors did survive.

2.7.2 Hill and Smith's "Evaluating Associativity in CPU Caches" [6]

This paper includes ideas for modeling caches and evaluating them with trace-driven simulation. Some of the important ideas (as viewed from 1998) are as follows: Section IIA reviews Mattson's *stack simulation*, which allows the simultaneous trace-driven simulation of alternative caches with the same number of sets. Section IVA formally presents *inclusion* and *set-refinement*, two useful properties for simultaneously simulating alternative caches. Section IVC presents *all-associativity simulation*, which enables the simultaneous simulation of alternative caches of all sizes and associativities (provided they use LRU replacement, have the same block size, and do no prefetching). Section VA presents the *3C model* and gives example data from an all-associativity simulation. Finally, Section VB reviews Smith's set-associative model and provides simulation data to support the hypothesis that set-associative caches miss more often than do fully associative ones because, due to random chance, some sets receive more active blocks than others.

2.7.3 Emer and Clark's "A Characterization of Processor Performance in the VAX-11/780" [4]

Even though the VAX-11/780 and the associated data are not current, Emer and Clark's paper illustrates an excellent example of how to perform system monitoring experiments and how the data obtained can be surprising and valuable. The VAX-11 architecture is a *complex instruction set computer* (CISC) with 32-bit virtual addresses, 16 general-purpose registers, no separate floating-point registers, and variable-length memory-to-memory instructions. Each instruction consists of a 1-byte opcode (that specifies the operation and number of operands) followed by 0–6 operand specifiers. Each specifier uses one of twenty-six methods to specify how to obtain an operand (e.g., register, various sizes of immediates, base plus displacement, indexed, and several more complex memory-referencing schemes). Specifiers also are of variable size with the *i + 1-st* specifier beginning after the *i-th* specifier ends. Thus, VAX-11 decoding is logically sequential both within and between instructions.

The paper begins by summarizing the VAX-11/780's implementation of the VAX-11 architecture (see Fig. 1). Control is specified with a microprogram and the microprogram program counter (uPC) that is exposed. The system monitoring data presented in this paper is obtained by attaching a probe to the uPC that records the number of times each uPC value was visited for up to 2 hours of execution time. The authors then use knowledge of the microprogram to translate uPCs into architectural events (that would happen on any VAX-11) and implementation events (VAX-11/780-specific). Experiments are run with two live nonrepeatable workloads (in vivo) and three synthetic repeatable ones (in vitro).

Architectural results show many things that are now widely known, in part, because of this paper. Simple instructions are common (84% of dynamic instructions). Branches are common (19%). Loop branches are usually taken (91%), whereas other conditional branches are less predictable (41% taken). Register operands are most common (41% of specifiers), whereas complex memory addressing modes are rare (7%). Average instruction size is 3.8 bytes. Instructions do 0.8 data reads and 0.4 data writes. (RISCs do fewer data memory operations because of more registers and better register allocation.)

Implementation also provided a wealth of information. The biggest surprise was that the VAX-11/780, widely known as a 1-million-instruction-per-second (MIPS) machine took 10.6 200-ns cycles to execute an average instruction. Thus, it was actually a 0.5 MIPS machine. Other data show that one half of execution time is spent decoding instructions, simple instructions use 10% of execution time (remember they are 84% of all instructions), and memory stalls waste 2.1 cycles per instruction (a small fraction of the VAX-11/780's 10.6 cycles per instruction, but a big deal for future RISCs that eliminated much of the VAX-11 decode overhead).

2.8 References

[1] A. Agarwal, M. Horowitz, and J. Hennessy, "An analytical cache model," *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," In *AFIPS Conference Proceedings*, pp. 483–485, Apr. 1967.

[3] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan Kaufmann, 1998.

[4] J. S. Emer and D. W. Clark, "A characterization of processor performance in the VAX-11/780," *Proceedings 11th International Symposium on Computer Architecture*, pp. 301–310, June 1984.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.

[6] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, C-38(12):1612–1630, 1989.

[7] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York: Wiley, 1991.

[8] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *The 17th Annual International Symposium on Computer Architecture*, pp. 364–373, May 1990.

[9] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[10] L. Kleinrock. *Queuing Systems Volume I: Theory*. New York: Wiley, 1975.

[11] L. Kleinrock. *Queuing Systems Volume II: Computer Applications*. New York: Wiley, 1976.

[12] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, N.J.: Prentice Hall, 1984.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, 9(2):78–117, 1970.

[14] J. R. Platt, "Strong inference," *Science*, 146(3642):347–353, 1964.

[15] S. Przybylski, M. Horowitz, and J. Hennessy, "Characteristics of performance-optimal multi-level cache hierarchies," *Proceedings 16th Annual International Symposium on Computer Architecture*, pp. 114–121, June 1989.

[16] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve, "The interaction of software prefetching with ILP processors in shared-memory systems," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 144–156, June 1997.

[17] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[18] A. J. Smith, "Cache memories" *ACM Computing Surveys*, 14(3):473–530, 1982.

[19] A. J. Smith, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Transactions on Software Engineering*, SE-4(2):121–130, 1978.

[20] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 380–391, June 1998.

[21] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *SIGPLAN Notices*, 29(6):196–205, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[22] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, 1965.