Methods for Performance Evaluation of Parallel Computer Systems

Yoram Eisenstadter

CUCS-246-86

Area Paper

Final Version

Yoram Eisenstadter

3 December 1986

Table of Contents

1. Introduction 1
2. Analytic Performance Modeling 2
    2.1. Introduction 2
    2.2. Queueing Models 3
    2.3. Problems in Analyzing Parallel Computers 4
    2.4. Advantages of Analytic Models 4
    2.5. Summary 5
3. Simulation Performance Modeling 5
    3.1. Introduction 5
    3.2. Stochastic Simulation 8
    3.3. Trace Driven Simulation 10
    3.4. Emulation 14
    3.5. Summary 17
4. Measurement 17
    4.1. Introduction 17
    4.2. Benchmarks 17
    4.3. Instrumentation 18
    4.4. Measurement of Emulated Architectures 21
    4.5. Model Validation 21
    4.6. Summary 22
5. Conclusions 23

1. Introduction

Although parallel computers have existed for many years, recently there has been a surge of academic, industrial and governmental interest in parallel computing. Commercially manufactured parallel computers have started to become available. Many new experimental parallel architectures are reported in the literature every year. Software for many types of applications, from scientific number crunching to artificial intelligence, is being written to run on parallel machines.

Performance is an essential consideration both in the design of new systems and the deployment of existing systems. Users of computers wish to utilize their hardware and software systems as efficiently as possible. Over the years, a field known as computer performance evaluation has arisen to address the problem of quantifying and predicting computer performance. Methods exist that can determine how efficiently a system's resources are being used. These can help track down the probable causes of performance problems.

There exists a large body of literature on computer performance evaluation in general. Most treatments of the field (see for example [Kobayashi 78], [Svobodova 76a], [Heidelberger 84]) agree that the work can be classified into three major categories:

1. *Analytic performance modeling:* mathematical techniques that yield the steady state (long term) behavior of systems modeled as queueing networks.
2. *Simulation performance modeling:* the writing of programs that mimic the behavior of the systems being modeled. These procedural models are more flexible and can express more detail than the mathematical models used in analytic performance modeling. There are three kinds of simulation:
    - *Stochastic simulation:* probability distributions describing the likelihood of various types of events are used to drive a model (usually a queueing model).
    - *Trace driven simulation:* a history, called a "trace," of actual events collected from a running program is used to drive a model of the system. The model reacts to the same sequence of events as the program did.
    - *Emulation:* programs written for the architecture being modeled are interpreted at the machine instruction level.
3. *Measurement:* empirical methods for assessing performance. Benchmark programs which exercise specific functions of a system can be timed. Hardware or software devices called instrumentation can be used to probe the state and record the actions of a running program, in order to determine its behavior and resource consumption.

This paper will survey the application of performance evaluation techniques to the study of parallel processing. We will be concerned primarily with research problems and not practical problems such as system performance tuning. The paper will emphasize simulation performance modeling, since we believe that this technique has the greatest potential for studying proposed parallel systems at a level of detail sufficient for understanding their intricacies. Measurement techniques can achieve the same level of detail, but are more expensive and cumbersome to apply. They also require the availability of the physical hardware being measured, which precludes the use of these techniques while an architecture is still being designed. We will show how analytical techniques break down when applied to typical problems in parallel processing.

An outline of the remainder of the paper follows. Section 2 takes a brief look at analytical performance evaluation methods. Section 3 presents detailed descriptions of simulation performance modeling methods, with numerous case studies of how they have been used to learn about parallel systems. Section 4 discusses instrumentation and benchmarking methods for the measurement of running programs, and how they can be used to validate performance models. The final section summarizes the most important results and techniques covered in this paper and speculates about future trends in performance evaluation of parallel systems.

2. Analytic Performance Modeling

2.1. Introduction

Analytic performance modeling is the use of mathematical techniques to solve systems of equations which express the steady-state behavior of computer systems. The systems are generally represented as queueing models in which processes are placed on queues when waiting for system resources. Although a general introduction to queueing theory is beyond the scope of this paper, we will explain how parallel computers or their subsystems can be represented as queueing models and what the limitations of this approach are. Understanding these limitations will motivate the design of the more flexible simulation models, which we cover in the next section.

2.2. Queueing Models

A computer system has various allocatable entities which are called resources. A typical resource might be a CPU or a disk drive. Various processes running on the system contend for the use of these resources. A common assumption in queueing models is that only one process may use a resource at a time, and that other processes requiring the use of that resource must wait their turn. In a queueing model, the resources are called servers and the waiting processes, called jobs, reside in queues.

Although the thought of multiple processes may bring to mind parallel processing, these models were first designed to represent uniprocessor systems running multitasking operating systems in which computation and I/O can be overlapped. Thus, a process *P1* waiting for the completion of some I/O operation would be suspended, and another process *P2* which was subsequently granted the CPU could request the same I/O device which was currently serving *P1*. Thus in order to proceed with its I/O, *P2* would have to wait for *P1* to release the I/O device. In the meantime, another process *P3* could be scheduled, etc.

Extensive work has been done using queueing models to predict the performance of multitasking systems under various scheduling strategies, or to optimize system configurations for maximal throughput. Such analytic techniques have been successfully employed to tune commercial computer systems. For example, the VM/370 Performance Predictor is a program used by IBM to analytically model the operation of the VM/370 operating system and the hardware it controls [Bard 78].

Systems with multiple processors have also been described using queueing models. However, these models tend to be more complex than uniprocessor models, and applying analytical solutions becomes more difficult. For example, [Browne 75] describes the analytic modeling of a system consisting of two CDC Cyber 70 CPUs and several disk and tape units. Although the analytical solution was a useful first order approximation, the model it used did not represent the system in as much detail as was ultimately required since it ignored the system's multiple peripheral processing units (I/O processors). The complete, detailed model proved to be mathematically intractable, and had to be solved by stochastic simulation methods.

2.3. Problems in Analyzing Parallel Computers

The bulk of the literature on analytic performance modeling deals with queueing networks which satisfy a condition called *product form*. The performance of such networks can be determined very economically, without explicitly solving for the probability of each possible state. Sauer and Chandy [Sauer 80] show, however, that many features generally found in queueing network models of parallel processors cause violations of product form. Examples of such features include simultaneous resource possession (e.g., a process holding a processor and memory simultaneously) and simultaneous job activities (e.g., a process spawning an independent subprocess). Queueing networks which are not in product form must be solved numerically, which for even modest networks, might involve solving a system of tens of thousands of linear equations. The only analytic method considered feasible for large networks not in product form is approximate numerical solution. The most common technique for doing this is *aggregation*, in which subnetworks of the model are replaced by single "composite" queues which approximate the flow through the subnetworks. This reduces the number of possible states of the system, resulting in a feasible numerical solution.

Using approximate models involves tradeoffs between the accuracy of the representation and the accuracy of the solution. On one hand, a model must include enough detail to convincingly account for the system behavior we wish to study; if a resource is ignored in a model, that model cannot be used to reason about that resource and its contributions to the overall behavior of the system. On the other hand, realistic models of parallel systems can only be solved analytically by employing approximations which introduce indeterminate errors into the solution. Analytical techniques have the additional drawback of only being able to model the steady state behavior of systems. Real systems exhibit frequent transitions between phases having different characteristic behaviors.

2.4. Advantages of Analytic Models

Although analytic models are severely restricted in the kind of constructs they can represent, they do have a tremendous cost advantage over simulation models. Once a queueing network model is solved mathematically, performance results can easily be re-evaluated for different

values of the input parameters (e.g., the average amount of time that a process uses a CPU can be varied). Simulations must be rerun in their entirety for each new set of parameters. Thus, whenever a problem can be adequately solved using analytic methods, such methods should be used.

Although analytic techniques may not be able to model entire parallel systems in sufficient detail, they may be employed to study individual components of these systems, such as interconnection networks or I/O subsystems. They can also be used to determine a quick first order approximation of a system's behavior.

2.5. Summary

Although analytical performance modeling techniques can provide mathematical solutions to many significant problems related to conventional multitasking systems, the modeling of true parallel computers by these means poses considerable problems. These problems arise because many of the characteristics of parallel systems violate the assumptions which allow exact solution of queueing networks, forcing solutions to be numerical approximations of unknown accuracy. Even numerical approximations can be expensive for complex systems having large numbers of states. It is therefore not surprising that there have been very few methods developed for the analytical solution of parallel computer systems [Heidelberger 84]. The next section discusses simulation models, which have the potential for much more flexibility, but at increased cost.

3. Simulation Performance Modeling

3.1. Introduction

Simulation performance modeling uses programs called simulators, whose behavior reflects the behavior of the systems being modeled. Simulation is more flexible than analytical performance analysis because the model can be represented by an arbitrary computer program rather than a set of equations which describe a queueing network model. Thus, simulations can be performed at almost any level of detail. Although simulations frequently do use queueing network models, the characteristics which make such models analytically intractable pose no significant obstacle

to simulation. Furthermore, although analytical models can only be used to study steady state behavior, simulation models can reflect dynamic behavior of systems. Thus, simulation is a powerful technique which can provide insights into the detailed behavior of parallel computing systems.

If simulation techniques can solve any queueing model, why bother with analytical techniques at all? The answer is that simulation techniques are much more computationally expensive than analytical techniques, making analytical techniques preferable when they can be used. Unfortunately, most of these techniques are not applicable to the detailed study of the behavior of parallel systems.

Simulators can be categorized based on how they represent the system's workload. Two major types of simulators are generally recognized: stochastic and trace driven. We find it useful to include emulators as a third category, although they can be thought of as variants of trace driven simulators. We will define these types of simulation in turn.

1) Stochastic simulation generally uses a random number generator with a given probability distribution to drive a queueing model. The statistics which are the output of the simulation reflect the steady-state behavior of the system. (In order to insure accurate results, data collected before the steady state is reached should be discarded.)

Since stochastic simulations are driven by random numbers, they are statistical experiments. Careful application of statistical techniques must be used to insure valid results. For example, successive runs of the same simulation may yield different results if the random number generator is given a different seed, or starting value. If the distributions driving these runs are, in the terminology of probability, independent and identically distributed (iid), conventional statistical techniques can be used to determine the expected values of the output parameters and their confidence intervals. Discussion of such statistical techniques is beyond the scope of this paper, and can be found in a simulation textbook such as [Mize 71].

2) Trace driven simulation: Although stochastic simulations can yield such overall performance statistics as throughput or response time, they cannot provide a detailed understanding of what

exactly happens when a real program is executed. For example, Smith points out that there exists no believable mathematical model of program behavior which accounts for phenomena that influence cache memory performance [Smith 85]. To analyze such problems, it is necessary to have a technique that is based on the behavior of real programs, not random number generators.

Trace driven simulation captures a sequence of events, or *trace*, from the actual execution of a program. This trace is then used as the workload model of a deterministic simulation of the system. Since the trace reflects the actual execution of some program, specific properties of that program can be discovered. A trace can be produced by running a program which is suitably instrumented to output trace records when significant events in the execution occur. For example, a parallel program may output trace data whenever a message is sent by one processor to another; such a trace can be examined to discover patterns of message traffic in that program. Once produced, the file containing the saved trace data can be used as input to a variety of analysis programs which examine the data in different ways or under a range of assumptions and parameters. To continue our example, the message trace file could be analyzed first assuming a communication model in which the cost of all messages was identical, and again under the assumption that certain communication paths are more expensive than others. Although the initial production of the trace may be very time-consuming, subsequent analyses do not require the replication of this work.

3) Emulation is the process whereby one computer performs a simulation of another computer. A program written for the emulated machine and executed on the emulator is a very detailed simulation of how the program would behave if actually run on the emulated architecture. Emulation is usually done by software programs, although emulators have been implemented in microcode for increased performance.

Having defined the various methodologies for simulation, we will now show how they have been used to study parallel processing systems.

3.2. Stochastic Simulation

This section presents some examples of how stochastic simulation has been used to model the behavior of parallel computer systems or their components. As we have already mentioned, stochastic simulations are appropriate when the information needed is a characterization of steady-state behavior of a system rather than details of its internal operation. We will see that stochastic simulations are also sometimes used to approximate more detailed deterministic simulations in order to improve simulation performance.

The most frequent use use of stochastic simulation in computer performance modeling is the modeling of system throughput as a function of CPU and I/O device parameters. Such systems are generally represented by a queueing network in which the CPU and various I/O devices each have queues to hold processes waiting to use them. Stochastic simulations are used to model the many kinds of queueing networks which cannot be feasibly solved using analytical methods. For example, Browne *et al.* used a stochastic simulation with a queueing model to study the performance of a computer system consisting of multiple CDC Cyber 70 mainframes, multiple I/O processors and about 100 disk drives [Browne 75].[1] Various distributions specified the paths of processes through the different parts of the queueing network, the amount of time they utilized system resources, and their priorities. The simulation was used to predict the performance of the system were it to be reconfigured to contain, for example, more powerful CPUs or a larger number of disk drives. This is a very frequent application of simulation in both research and production environments.

Stochastic simulation has also been used extensively to study the performance of interconnection networks. Such networks, which are used to connect processors and memories together in many tightly coupled parallel computers, are of great interest since they are frequently a factor which limits performance. Some types of interconnection networks are very costly to construct. Thus, it is important to be able to simulate what the communication requirements of a parallel processor will be before designing its network. As with the previous examples, the important point is that by using simulation, a space of design options can be explored and potential problems can be uncovered without having to build and measure actual hardware.

---

[1]The analytical modeling of a portion of this system was presented in the previous section.

Reed used stochastic simulation to study which classes of interconnection network best supported various types of workloads [Reed 84]. Two types of workloads he considered were static assignments of large tasks to processors and dynamic creation of small tasks at run-time. He modeled the workloads using probability distributions to determine parameters such as task running time and task creation time. He simulated five types of interconnection networks, and determined that a designer's choice of network topology should be dependent on the types of workloads that the machine was intended to process.

Snir wrote a program called NETSIM [Snir 81] to simulate the interconnection network of the NYU Ultracomputer [Gottlieb 83], a shared-memory parallel processor. Because a detailed functional simulation of this network was deemed to be too expensive for everyday use, stochastic methods were used to approximate network behavior. The network used in the Ultracomputer is a buffered Omega network, which has queues at each of its switch points to buffer packets of data waiting to travel to the next stage of the network. The simulation is based on a queueing system model in which the probability of a packet moving from one stage to another is given by a theoretically derived distribution. This distribution is based on the assumption that memory references are uniformly distributed among the system's memory modules. The results of the simulation were in general agreement with those produced by a detailed deterministic simulation.

Pfister and Norton [Pfister 82a] simulated the interconnection network of RP3, an MIMD parallel computer under construction at IBM [Pfister 85]. The purpose of their study was to determine whether their network should support message combining, the merging of requests destined for the same address; if a non-combining network could perform acceptably well, it would be much cheaper to build than a combining network. The network they simulated was a variant of the design used for the NYU Ultracomputer. They constructed a deterministic simulation of the proposed networks, and subjected them to a sequence of requests whose addresses were uniformly distributed with the exception of a single spike, which they called a "hot spot," at one particular address; this corresponded to a frequently accessed global lock variable. The non-combining network suffered significant global performance degradation as the result of the hot spot, whereas the combining network did not. As a result of these simulations, they decided to build a combining network.

[Hillyer 86] studied the processing of the relational database join algorithm on the NON-VON 4 architecture [Shaw 84]. His goal was to determine whether the bottleneck in the algorithm was disk performance or interconnection network bandwidth. While the network simulations described above aimed at understanding properties of the interconnection networks themselves, Hillyer's work used a network simulation along with a disk simulation to understand a particular algorithm. The network was simulated with a very detailed and accurate deterministic algorithm. The disk drives, one of which was attached to each processing element of the NON-VON 4 machine, were simulated using a stochastic simulation in which the rotational delays and track seek times of the disks were determined by a uniform distribution. The two simulators were interfaced to each other so as to model the effects of reading database records and sending their contents to other processing elements by means of the interconnection network, the destinations being uniformly distributed to simulate hashing on the database keys of the records. Hillyer finally determined that the disk transfer rate was the bottleneck in the system.

Examination of the work we have presented on stochastic simulation suggests that in the modeling of parallel systems, the expressive power of simulation models is indeed superior to that of analytical models. We are now starting to see models which include limited knowledge about the behavior of the software as well as the hardware. For example, Reed's work examined different kinds of process creation patterns, and Hillyer's studied the dynamics of a particular algorithm. Even more detailed information about software behavior can be captured by trace driven simulation, which is the next topic.

3.3. Trace Driven Simulation

Trace driven simulation, which was first considered in [Cheng 69], has been used to study many different aspects of computer system behavior. The earliest uses of this technique were for studying operating system scheduling algorithms. Later, it was used to evaluate the performance of demand paging algorithms. Smith maintains that trace driven simulation has been used in almost every research paper which analyzes cache memory performance [Smith 85]. Communication in distributed systems has been analyzed by examining traces of message-passing events. Trace driven simulation has also been extensively utilized to measure many of these effects in parallel computer systems.

Some fundamental properties of trace driven simulation are illustrated by Sherman, who describes the use of event traces to study operating system scheduling policies [Sherman 72]. A trace was produced by instrumenting an operating system to collect records of CPU usage duration and I/O service times. This trace was then run through a system model under each of the various scheduling algorithms to determine their relative merits. The trace provided a reproducible series of demands on the system; previous work had merely changed the scheduling algorithms, and required a great deal of analysis to account for the different job streams under which the different algorithms were tested. The authors concluded that trace driven modeling was "an excellent vehicle for performing controlled scientific experiments to evaluate resource allocation policies in computer systems."

More recently, trace driven simulations have been used to evaluate the performance of parallel computer hardware. We will consider two parallel computer systems: RP3 and Cedar.

The RP3 computer [Pfister 85] consists of several powerful processing units among which global memory is shared by routing memory references through a log-stage interconnection network. [So 86] describes the software system, called PSIMUL, used to simulate the RP3. Trace driven simulation was used to simulate RP3's cache and network. Memory reference traces (i.e., files containing one record specifying each memory reference made by the parallel program) are produced from parallel programs running under an emulation program called SEMUL. These traces are run through a cache simulation model, which allows the user to vary certain cache parameters, such as the size of the cache. This simulation yields statistics about cache performance, e.g. the percentage of references which were satisfied from the cache (called cache hits). Furthermore, it produces a new trace which contains only cache misses. This trace can be used to drive a simulation model of the interconnection network, which reports the network utilization and total running time of the parallel program.

PSIMUL is notable for its ability to exploit multiple real CPUs, if available, to speed up the simulation process. For example, a simulation running on an IBM 3081, a dual CPU system, simulating an eight CPU parallel machine, produced a trace of over 150 million instructions in about one hour. Sophisticated buffering schemes are used to increase the I/O throughput of the

simulation. Other researchers have also discovered the utility of using multiple CPUs to speed up the computationally demanding task of simulation. See, for example, [Misra 86].

Trace driven simulation has also been used extensively in the design of the Cedar parallel computer [Abu-Sufah 85]. A program whose execution on Cedar is to be simulated is first run through a restructuring program which automatically generates parallel code. The restructured program is then run through a trace generator, TRGEN, which extracts traces from it. Information recorded in the trace file includes global memory references and process creations. The traces are used as input into a variety of simulation models, including a global memory simulator (MEMSIM) and a global interconnection network simulator (NETSIM). These simulators produce a variety of statistics about the performance of the parallel program.

The preceding examples illustrated the use of general purpose simulators which could determine the performance characteristics of an arbitrary program running on some simulated machine. We now turn our attention to more specialized work which uses trace driven simulation to study in detail the performance of specific software systems. We present simulation examples of three AI applications running on parallel computers and of an object oriented system running on a conventional machine.

[Fennell 75] and [Fennell 77] describe the simulation of Hearsay II, a speech understanding system designed to run on parallel machines such as C.mmp [Wulf 72]. A multiprocessor simulator running on a DECsystem-10 collected detailed trace information about calls to system service routines (such as those for synchronization and process creation). The traces were post-processed by a collection of programs which determined multiprocessing overhead, degree of parallelism and interference between processors.

Further work on C.mmp was done by McCracken, who built a version of Hearsay called HSP which used a production system as its control paradigm [McCracken 81]. Since McCracken anticipated that HSP would have more parallelism than could be exploited by the then existing 16 processor C.mmp prototype, he developed a trace driven simulator which, given trace data from HSP running under C.mmp produced timing projections for a 50 to 100 processor machine. The simulation captures enough detail to account for short critical sections in the code and for processor idle time at the end of each production system cycle.

[Miranker 86] studied the execution of OPS5 production systems on DADO [Stolfo 83], a tree structured parallel computer. Due to hardware limitations of the DADO machine, he had to derive the performance characteristics of the parallel execution for some algorithms by simulation. (Other algorithms were executed directly on DADO and timed.) To do this, he instrumented a serial version of his production system interpreter to produce traces containing detailed information about its operation. The trace file was fed into a postprocessing program which, given the number of available DADO processing elements and a partitioning of the production rules among processing elements, simulated the parallel execution of the traced production system on DADO, and provided an estimate of what the execution time on DADO would be.

Miranker reports that the traces produced for his research have been useful to other researchers as well. The traces were used for preliminary studies of the HerbAl set oriented production system language [vanBiema 86] by modifying them to reflect the state of the execution had the production rules been written in HerbAl rather than OPS5. The modified traces were then run through the simulator as described above. In fact, it is frequently the case that trace data, once gathered, can be used repeatedly for a large variety of different applications.

Zorn, who is studying garbage collection algorithms for parallel Lisp programs [Zorn 86], uses trace driven simulation to model the performance of those algorithms. Lisp programs are instrumented to produce traces containing information about the creation of objects and references to objects. A variety of garbage collection algorithms can be compared by writing simulators for the algorithms, and running traces of benchmark programs through them.

Stamos studied the effects of various schemes for the placement of Smalltalk objects in virtual memory systems [Stamos 84]. His traces were produced by a novel method: instead of recording the virtual or physical addresses of the referenced objects as was done by previous researchers who studied virtual memory, he recorded references to the objects in a symbolic form, which included the name of the object and the field of the object that was referenced. This trace, along with a memory map which gave the address of each object, was used as input into a variety of simulators for different memory models, including a conventional demand paged virtual memory

and an object-oriented memory. Because the reference trace contained no addresses, different initial placements of the objects could be simulated using the same trace; only a new memory map was required. Stamos also used a trace compression algorithm which removed sequential references to objects on the same page. This resulted in improved performance of the simulation, and preserved the original sequence of page faults.

3.4. Emulation

Emulation methods, which interpret programs at the machine instruction level, have been used quite successfully in the development of parallel hardware and software. While trace driven simulations usually postprocess previously collected trace files that reflect certain events in some program's execution, emulation closely resembles the actual execution of programs and can thus provide even more detailed information about the software, the hardware and their interactions. Some researchers have actually built small operating systems to run on their emulators, and have found these emulated environments so useful for debugging and measurement that they continue to use them even after they have working hardware prototypes available. The chief drawback to emulation is the massive amount of CPU power required to perform emulation at speeds high enough to support the emulation of non-trivial programs.

Two decades ago, in a survey of parallel processing hardware and software, Lehman advocated the use of simulation as a design tool for parallel processors [Lehman 66]. He described his own work on an emulation program, which he called an executing simulator, that modeled a parallel processor, interleaved memory modules and a processor-memory switch. The simulated processing elements were modeled after the IBM 360 and executed IBM 360 machine language augmented with special multiprocessing instructions. Lehman emulated the execution of numerical analysis programs, studying their performance as more memory modules were made available to them. Statistics about instruction execution as well as overall program execution were maintained. For example, it was determined that 67 percent of memory fetches were for instructions and the remaining 33 percent for data.

Svobodova and Mattson built a parallel microcoded emulation machine called the MMP from seven CDC-5600 series processors [Svobodova 76b]. This machine was specifically designed

for the high-speed emulation of other computer architectures. Instructions of the emulated machine were described by coding them in MMP microcode. The execution environment included extensive measurement and debugging facilities. The application described was the emulation of the U.S. Army TACFILE system, including an AN/GYK-12 processor and its assorted peripherals. More recently, a number of other microcoded emulation engines have been built. The use of such a machine to emulate the BELLMAC-32A microprocessor is described by [Salomon 82], who reports that such an emulator was used to provide a software development environment for the microprocessor before the real hardware was available. The emulation microcode was heavily instrumented to provide performance monitoring and debugging facilities. Commercial microcoded emulation engines have been produced by such companies as Microdata and Nanodata.

The WASHCLOTH parallel program simulator [Gottlieb 80a] is used to develop programs for the NYU Ultracomputer [Gottlieb 83]. WASHCLOTH runs on a CDC-6600 computer; it interprets CDC-6600 instructions produced by standard CDC compilers. The instructions are executed on a round robin basis, one instruction from each simulated processor in turn. The original WASHCLOTH program simulated a paracomputer [Gottlieb 83] rather than an Ultracomputer,[2] a paracomputer being an idealized machine in which memory can be simultaneously read and written by multiple processors without contention. Various additions to the software, such as the NETSIM network simulator (which we discussed in Section 3.2), allowed for more realistic simulations of the Ultracomputer hardware. WASHCLOTH was used as a parallel program development environment before the Ultracomputer hardware was operational, and is still used because of its flexibility. A rudimentary operating system called MOP has been implemented on top of WASHCLOTH [Gottlieb 80b]. MOP has primitives for allocation, deallocation and suspension of processors, but doesn't support processor preemption or dynamic process creation.

The PSIMUL [So 86] simulation environment for the development of the RP3 parallel computer

---

[2]The Ultracomputer is actually built from Motorola 68000 microprocessors, not the CDC-6600 processors which WASHCLOTH simulates. This introduces some degree of inaccuracy.

provides two emulation methods. The underlying method is VM/EPEX, an environment for executing parallel programs running on multiple virtual IBM 370 machines created by the VM/370 operating system. VM/EPEX provides shared memory and various synchronization primitives. The parallel programs running under VM/EPEX can use a facility called SEMUL, which interprets the opcodes in a specified subroutine. SEMUL is the program which actually produces the reference traces which we described in the previous section. SEMUL can interpret IBM 370 machine instructions produced by normal IBM compilers. It collects extensive execution statistics including total instruction counts, instruction counts by opcode and number of memory references. Although these tools emulate IBM 370 processors rather than the ROMP processors used in the actual RP3 prototype, the instruction traces they produce can be run through RP3 network and cache simulators. The tools provide a convenient and powerful environment in which parallel programs can be developed.

The PARSIM simulation facility supports various types of parallel program execution [Board 83]. The architecture which PARSIM models is specified by naming the functional units and specifying the connections between them. A parallel program emulator is provided, as well as facilities for executed uninterpreted instruction streams and statistically generated instruction streams. PARSIM also provides process creation and interprocess communication facilities.

[Lieberman 83] simulated a proposed parallel, object-oriented architecture called the Apiary [Hewitt 80]. He represented processing elements and connections between them as objects, using the Flavors facility [Moon 86] on a Lisp Machine. A processing element object would simulate one primitive machine operation upon receiving a message called *TICK*. The simulator was very flexible, allowing any object to be replaced with one that was instrumented for debugging or measurement, so long as it had the same message passing behavior. The entire machine could be single-stepped by manually issuing *TICK* messages. A window-oriented machine language debugger was also written.

3.5. Summary

We have surveyed the three major types of simulators: stochastic simulators, trace driven simulators and emulators. We have shown several applications of each of these techniques. These types of simulators were presented in order of increasing level of model detail. The models ranged from simple queueing models with randomly generated inputs to detailed emulations of parallel programs at the machine instruction level.

4. Measurement

4.1. Introduction

Measuring the running time of an program on a particular machine is the most obvious way to quantify performance. However, there are many statistics about a program's execution that we would like to obtain, which can not be obtained using only a stopwatch. In some cases, we can not even run the program on the intended architecture since the architecture has not yet been built but still wish to measure the performance of that program on that architecture. Accurate emulation techniques can provide such performance data. Measurement is, however, the most robust way to validate the results of analytical or simulation models. Validation, or the measurement of the accuracy of models, will be discussed in detail.

4.2. Benchmarks

Perhaps the most straightforward way of determining how a system behaves is to directly measure it. For example, one could run a typical program, or *benchmark*, on a given machine and time its execution. However, the only data this exercise will ultimately provide is the running time of the program. Careful experimental design can yield further insight. For example, knowing exactly what resources our benchmark consumes, and the running times for the identical benchmark program on other architectures allows us to deduce which particular features of our machine could account for the performance patterns we observed. An excellent illustration of this "art" of benchmarking is [Gabriel 85], in which the author devised a collection of benchmarks to test various important performance aspects of Lisp systems, and compared their performance on several different machines. Gabriel's benchmarks have become widely adopted as a standard for evaluating Lisp compilers and architectures, to the extent that

manufacturers will tune their Lisp systems specifically to get high performance on those benchmarks.

Benchmarking is probably the most common technique used to evaluate the relative performance of hardware and software products. Related techniques include *kernels* and *synthetic programs* [Bauer 77]. Kernels are representative programs whose exact execution parameters are known (e.g., a kernel may be designed to read 10,000 records of a given size from a disk file), but whose implementation details (e.g., the exact operating system calls used to read the file) are left to be optimized for a given system. Kernels can potentially give more accurate information about a particular machine than benchmarks, since they don't rely on a program which may have been written with some particular system in mind. *Synthetic programs* are programs that don't perform any specific function, but are used to exercise all possible functions of a piece of hardware, sometimes under extreme conditions.

Although benchmarks and the related techniques discussed above are very useful in many situations, they have major drawbacks which have motivated the design of alternate measurement techniques. A series of benchmarks which exercise a variety of system features can be run to identify general areas of good and poor performance, but the only quantitative results which are produced are the running times of the benchmarks. Such results cannot explain *why* certain operations are efficient and others are not. To understand why a system performs the way it does, it is necessary to understand it at a finer level of detail. For example, we may wish to determine what percentage of the memory references in a parallel program are to shared data, and how much synchronization overhead is associated with these references. Thus, we now turn our attention to more detailed measurement techniques.

4.3. Instrumentation

Both hardware and software can be equipped with measuring devices which monitor and collect information about significant events or parameters. Such devices are known collectively as *instrumentation*. Hardware and software monitoring can be employed separately, but since each of these methods has its own set of advantages and limitations, they are frequently used together. In the section on simulation we mentioned that trace driven techniques used data collected from

running systems to drive simulations. The instrumentation techniques described in this section are the means whereby such data is collected.

1) Hardware monitors are electronic devices which can be attached to computers in order to collect operational data. A monitor has a number of high-impedance connectors, called probes, which are attached at significant locations, or probe points, in the circuitry of the computer. These probes send information back to the hardware monitor, which typically contains the high-speed logic necessary to capture this information in real time, as well a small computer to control the acquisition, recording and reporting of the data. For some commercial computers, libraries of probe points for observing a variety of interesting signals are available from the manufacturer; for custom designed prototypes, the designer can locate the probe points. The advent of VLSI microprocessors, however, has made it increasingly difficult to gain access to signals which divulge a processor's internal state.

Hardware monitors can record such events as context switches, page faults, cache hits, I/O operations to selected channels or devices, execution of specific instructions, suspension of CPU activity to wait for I/O, loading of selected registers, etc. They generally do not have access to knowledge of what is occurring at the software level. For example, they cannot know which file is being read from a disk or the name of the subroutine currently executing.

The greatest advantage of hardware monitors is that they are non-intrusive. Use of the monitor does not degrade system performance, as contrasted with the execution of software monitor code. Hardware monitors can also record events such as cache memory hits, which are inaccessible to software monitors. Their disadvantages include the high cost of the additional monitoring hardware, the inconvenience of locating probe points and physically attaching probes to a computer, and the inability to access software information. The practice of hybrid monitoring, or using hardware and software monitors concurrently, compensates for the hardware monitor's inability to sense software events. In fact, most recent uses of hardware monitors have been in hybrid monitoring situations.

Although the literature abounds with references to hardware monitoring studies, very few of these studies have had parallel computers as their object. One exception is [Fromm 83], which

describes the use of hardware and software monitors with the Erlangen General Purpose Array, a tightly coupled parallel computer. A hardware monitor called Zahlmonitor III collects trace data from each of the processing elements, merging it into a single trace. The software monitor places process IDs into a special register, where they can be read by the hardware monitor.

2) Software monitors collect information about a system by means of software probes, which are sections of code added to a program to collect trace data and statistics pertaining to the program's execution. Many large programs, especially operating systems, have monitor code built into them to facilitate the gathering of performance data. Examples are IBM's VM/370 [Bard 78] and the ARPANET interface processor (IMP) control programs [Kleinrock 74]. Other programs have been modified to incorporate software probes.

One advantage of using software monitors is their flexibility. Whereas hardware probes are difficult to relocate, a new software probe can be added by simply modifying and recompiling the code. Furthermore, software monitors have access to high-level software information unavailable to hardware monitors; the inclusion of such information in monitor trace output makes the output much easier to analyze. The only major disadvantage of software monitors is that they are intrusive. Since the monitoring instructions inserted in the programs take time to execute, software monitoring degrades system performance, sometimes significantly. This overhead must be accounted for to improve the accuracy of the measurements.

We have already seen extensive examples of the utilization of trace data from software monitors in the section on trace driven simulation. Thus, we will present only one additional case study here, with emphasis on the process of software instrumentation itself, and not its results.

[Dritz 86] describes the performance evaluation of simple parallel Lisp programs running on the Encore Multimax and the Denelcor HEP, two commercial parallel processors. The initial instrumentation consisted of a single software probe to measure the time which processes spent acquiring locks for shared data structures. The insights from this experiment led to refinement of both the algorithms (to reduce lock contention) and the monitoring code (to reduce monitoring overhead). Software probes for monitoring processor idle time were subsequently added. The inherent flexibility of software instrumentation allowed it to be used incrementally, in an exploratory manner.

4.4. Measurement of Emulated Architectures

Simulators which perform software emulation of parallel machines were previously surveyed. Such emulators are usually extensively instrumented to collect performance data, and can be a substitute for measurement of the real hardware if the hardware is not yet operational. The accuracy of the performance data is a function of the level of detail of the emulation.

Some emulation programs, such as WASHCLOTH for the NYU Ultracomputer and PSIMUL for the IBM RP3 can provide only approximate timing data, since the processors they emulate are not the processors of the actual target architectures (recall that WASHCLOTH emulates CDC-6600 processors and PSIMUL emulates IBM 370 processors).

Generally, microcoded emulation machines collect more accurate timing information. For example, the MMP machine, described in the emulation section, maintains a virtual clock which accurately reflects execution time on the emulated architecture. This allows true performance measurements of the emulated architecture to be made.

4.5. Model Validation

Measurement is frequently used to check the results of analytical or simulation models for accuracy. Since a model is an abstraction of some real system, it doesn't perform all of the system's functions. Validation is the process of determining whether the model accurately reflects the behavior of the system it was designed to represent. This is usually done by using the model to predict the performance of a configuration of the system whose performance can also be directly measured. If significant differences exist between the model and the real system, the model must be corrected. [Rose 77] is an example of empirical validation of a analytical model.

Measurements of real systems can also be the initial sources of various parameters for a simulation or analysis [Rose 78]. For example, the average time a process waits for a disk drive can be measured by hardware or software monitoring. This value can then be used to specify the corresponding parameter in an abstract model of a disk drive. Increasing the accuracy of the input parameters of the model will make its predictions more reliable. Thus, it is desirable to

check the assumed parameters of the system model against the real system or a similar one. A good example of interaction between analytical models and measurement is the VM/370 Performance Predictor [Bard 78], which uses a software monitor built into VM/370 to gather data used by the analytical model.

If a real system is not available for validation measurements, the best alternative is to check the model against a more detailed model. Emulation models are good choices, since they capture a large amount of the detail of the real hardware. If an emulator is not available, comparing the results of two different types of models, for example an analytical solution and stochastic simulation results, could help. If the results disagree, at least one of the models is defective. If the results agree, there is a higher probability that both are reasonable. [Browne 75], for example, compared the predictions of an analytical model with those of a more detailed stochastic simulation and found them to be similar. [Lazos 81] studied the validation of models by other models, and concluded that models could differ quite significantly from the real system and still produce fairly similar results. This suggests that even fairly simple models could be used for quick validations in situations where very high accuracy is not required.

4.6. Summary

A variety of methods for the performance measurement of parallel computers have been presented. These include benchmarking, hardware and software monitoring, and use of emulation software or hardware. In many cases the decision as to which method to employ is quite easy. For example, if no hardware implementation of an architecture exists, one can only measure the performance of a simulated machine. If the probe points for a machine cannot be determined, then it is impossible to use a hardware monitor. In the remaining cases, the decision is driven by the cost of the measurement method and the degree of accuracy required. Software instrumentation is usually more economical and flexible than hardware instrumentation, but not as accurate since it imposes monitoring overhead on the machine.

Measurement is frequently used to validate the results of analytical or simulation models. Validation serves a crucial role in performance modeling, since without actually comparing the projections of models against measured results it is impossible to know how trustworthy the

models are. Sometimes, when measurement of the real hardware is not possible, increased confidence in a model can result from comparing it with a different class of model.

5. Conclusions

This paper has surveyed the major classes of computer performance evaluation techniques and shown how they have been used to study parallel computer systems. Simulation plays a much greater role in this context than do analytical techniques because of the flexibility and expressiveness of simulation models. Emulation techniques, although relatively expensive to use, seem to be the method of choice for achieving a detailed understanding about the operation of parallel software. A table summarizing the work we presented appears at the end of this section.

One big limitation of detailed simulation is the huge amount of CPU time required. Increased use of parallel machines to simulate other parallel machines may speed up this process by orders of magnitude. Such simulation facilities would provide the ability to do more significant simulation work prior to designing new architectures. Currently, only small pieces of software can be exhaustively analyzed. Perhaps in the future, the performance of large software systems on a proposed architecture could be evaluated without actually implementing the architecture. This would significantly decrease the costs and increase the productivity of parallel hardware and software research. Various parallel architectures for simulation have already been built, some of them very powerful. We have already mentioned certain microcoded emulation machines. Another interesting architecture is IBM's Yorktown Simulation Engine (YSE), a massively parallel machine used for gate level simulation of logic [Pfister 82b]. It has been estimated that it could emulate an IBM 3081 computer at the rate of 1000 instructions per second, and some commercial microprocessors at rates faster than their normal execution. The YSE also has extensive diagnostic and trace facilities built into its software. Such machines may prove useful for the simulation of parallel architectures.

Innovative software techniques may make the job of parallel computer performance evaluation easier. For example, [Ishida 75] built a graphical tool for monitoring the execution of a four CPU multiprocessor. [Steinberg 86] describes a graphical interface which monitors execution of

parallel Lisp programs on the BBN Butterfly parallel computer. Such techniques could enable researchers to better understand the behavior of systems in which many events are happening simultaneously.

Performance evaluation techniques have been, and will probably continue to be, an integral part of the design process for parallel machines. Advances in hardware and software technology will make possible the performance evaluation of increasingly larger systems with greater ease and efficiency.

Table 5-1: Summary of Work Surveyed

Reference	What Studied	Methods Used[3]	Comments
Bard 78	VM/370 operating system performance	APM, SMON	Used monitor data as input to simulation
Browne 75	2 CDC Cyber CPUs and peripherals	APM, SSIM	APM couldn't represent I/O processors
Sauer 80	Queueing networks in general	APM	Approximate solutions to queuing models
Reed 84	Various interconnection topologies	SSIM	
Snir 81	Ultracomputer interconnection network	SSIM	
Pfister 82a	RP3 interconnection network	SSIM	Found hot-spot phenomenon
Hillyer 86	Relational joins on NON-VON 4	SSIM	
Smith 85	Cache memories in general	TDSIM	TDSIM used for most studies of caches
Sherman 72	O.S. scheduling policies	TDSIM	
So 86	Execution of programs on RP3	TDSIM	Used PSIMUL program
Abu-Sufah 85	Execution of programs on Cedar	TDSIM	
Fennell 75	Hearsay II on C.mmp	TDSIM	
McCracken 81	HSP on C.mmp	TDSIM	
Miranker 86	Productions systems on DADO	TDSIM	
van Biema 86	Production systems on DADO	TDSIM	
Zorn 86	Parallel garbage collection	TDSIM	
Stamos 84	Paging in object oriented systems	TDSIM	Trace compaction sped up simulation
Lehman 66	Execution of parallel programs	EMUL	
Svobodova 76b	Execution of TACFILE system	EMUL	Microcoded emulation engine
Salomon 82	BELLMAC-32A microprocessor	EMUL	Microcoded emulation engine
Gottlieb 80a	Execution of Ultracomputer programs	EMUL	
So 86	Execution of RP3 programs	EMUL	
Board 83	Parallel program execution	EMUL, SSIM	User describes architecture
Lieberman 83	Apiary object-oriented machine	EMUL	
Gabriel 85	Lisp programs on uniprocessors	BENCH	These benchmarks now widely adopted
Fromm 83	Erlangen General Purpose Array	HMON	
Kleinrock 74	ARPANET control program	SMON	
Dritz 86	Execution of parallel Lisp programs	SMON	
Lazos 81	Validation of models using other models		
Misra 86	Distributed simulation algorithms		

[3]APM = Analytic performance modeling, SSIM = Stochastic simulation, TDSIM = Trace driven simulation, EMUL = Emulation, BENCH = Benchmarking or related technique, HMON = Hardware monitor, SMON = Software monitor.

References

[Abu-Sufah 85] Abu-Sufah, W. and A.Y. Kwok.
Performance Prediction Tools for Cedar: a Multiprocessor Supercomputer.
In *Proc. 12th Annual International Symposium on Computer Architecture.* 1985.

[Bard 78] Bard, Y.
The VM/370 Performance Predictor.
ACM Computing Surveys 10(3):334-342, Sept, 1978.

[Bauer 77] Bauer, F.L. *et al.*
Software Engineering.
Springler-Verlag, 1977.

[Board 83] Board, J.A. and P.N. Marinos.
An Interactive Simulation Facility for the Evaluation of Shared-Resource Architectures.
In *20th Design Automation Conference.* IEEE, 1983.

[Browne 75] Browne, J.C., *et al.*
Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems.
In *Proceedings of the IEEE.* 1975.

[Cheng 69] Cheng, P.S.
Trace-driven system modeling.
IBM Systems Journal 8(4), 1969.

[Dritz 86] Dritz, K.W. and J.M. Boyle.
Beyond Speedup: Performance Analysis of Parallel Programs.
1986.
Unpublished draft from author at Argonne National Laboratory.

[Fennell 75] Fennell, R.D.
Multiprocess Software Architecture for AI Problem Solving.
PhD thesis, Carnegie-Mellon University, May, 1975.

[Fennell 77] Fennell, R.D. and V.R. Lesser.
Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II.
IEEE Transactions on Computers C-26(2), Feb, 1977.

[Fromm 83] Fromm, H., *et al.*
Experiences with performance measurement and modeling of a processor array.
IEEE Transactions on Computers C-32(1), January, 1983.

[Gabriel 85] Gabriel, R.P.
Performance and Evaluation of LISP Systems.
MIT Press, 1985.

[Gottlieb 80a] Gottlieb, A.
WASHCLOTH - The Logical Successor to Soapsuds.
Technical Report Ultracomputer Note No. 12, New York University, December, 1980.

[Gottlieb 80b] Gottlieb, A.
MOP - A (Minimal) Multiprocessor Operating System Extending WASHCLOTH.
Technical Report Ultracomputer Note No. 13, New York University, December, 1980.

[Gottlieb 83] Gottlieb, A. *et al.*
The NYU Ultraccomputer -- Designing an MIMD Shared Memory Parallel Computer.
IEEE Transactions on Computers C-32(2), Feb, 1983.

[Heidelberger 84] Heidelberger, P. and S.S. Lavenberg.
Computer Performance Evaluation Methodology.
IEEE Transactions on Computers C-33(12), December, 1984.

[Hewitt 80] Hewitt, C.
The Apiary Network Architecture for Knowledgeable Systems.
In *Proc. 1st Lisp Conference, Stanford University.* 1980.

[Hillyer 86] Hillyer, B.
On Applying Heterogeneous Parallelism to Elements of Knowledge-Based Data Management.
PhD thesis, Columbia University, 1986.

[Ishida 75] Ishida, H., S. Nomoto and H. Ozawa.
Graphic Monitoring of the Performance of a Large 4-CPU Multiprocessor System.
In *USA/Japan Conference Proceedings.* 1975.

[Kleinrock 74] Kleinrock, L. and W. Naylor.
On Measured Behavior of an ARPA Network.
In *National Computer Conference.* AFIPS, 1974.

[Kobayashi 78] Kobayashi, H.
Modeling and Analysis: An Introduction to System Performance Evaluation Methodology.
Addison-Wesley, 1978.

[Lazos 81] Lazos, C.
Comparison of Simulation Results of Dissimilar Models.
In *14th Annual Simulation Symposium.* 1981.

[Lehman 66] Lehman, M.
A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors.
In *Proceedings of the IEEE*, pages 1889-1901. December, 1966.
Also in Bell, C.G., *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[Lieberman 83] Lieberman, H.
An Object-Oriented Simulator for the Apiary.
In *Proc. AAAI-83*. American Association for Artificial Intelligence, Aug, 1983.

[McCracken 81] McCracken, D.L.
A Production System Version of the Hearsay-II Speech Understanding System.
UMI Research Press, 1981.
CMU PhD Thesis.

[Miranker 86] Miranker, D.
TREAT: A New and Efficient Match Algorithm for AI Production Systems.
PhD thesis, Columbia University, 1986.

[Misra 86] Misra, J.
Distributed Discrete-Event Simulation.
ACM Computing Surveys 18(1):39-66, March, 1986.

[Mize 71] Mize, J.H. and J.G. Cox.
Essentials of Simulation.
Prentice-Hall, 1971.

[Moon 86] Moon, D.A.
Object-Oriented Programming with *Flavors.*
In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings.* ACM, Sept., 1986.
Published as *SIGPLAN Notices* Vol. 21, No. 11, Nov. 1986.

[Pfister 82a] Pfister G.F. and V.A. Norton.
Hot Spot Contention and Combining in Multistage Interconnection Networks.
IEEE Transactions on Computers (10), October, 1982.

[Pfister 82b] Pfister, G.F.
The Yorktown Simulation Engine: Introduction.
In *19th Design Automation Conference.* IEEE, 1982.

[Pfister 85] Pfister, G.F., *et al.*
The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.
In *International Conference on Parallel Processing.* IEEE, 1985.

[Reed 84] Reed, D.A.
The Performance of Multimicrocomputer Networks Supporting Dynamic Workloads.
IEEE Transactions on Computers C-33(11):1045-1048, November, 1984.

[Rose 77] Rose, C.A.
A ''Calibration-prediction'' Technique for Estimating Computer Performance.
In *National Computer Conference.* AFIPS, 1977.

[Rose 78] Rose, C.A.
A Measurement Procedure for Queuing Models of Computer Systems.
ACM Computing Surveys 10(3):263-280, Sept, 1978.

[Salomon 82] Salomon, F.A. and D.A. Tafuri.
Emulation -- A Useful Tool in the Development of Computer Systems.
In *Proc. 15th Annual Simulation Symposium.* 1982.

[Sauer 80] Sauer, C.H. and K.M. Chandy.
Approximate Solution of Queuing Models.
IEEE Computer 13(4), April, 1980.

[Shaw 84] Shaw, D.E.
SIMD and MSIMD Variants of the NON-VON Supercomputer.
In *Proceedings of the COMPCON Spring '84.* February, 1984.

[Sherman 72] Sherman, S., F. Basket III and J.C. Brown.
Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System.
Comm. ACM 15(12), December, 1972.

[Smith 85] Smith, A.J.
Cache Evaluation and the Impact of Workload Choice.
In *Proc. 12th Annual International Symposium on Computer Architecture.* 1985.

[Snir 81] Snir, M.
NETSIM Network Simulator for the Ultracomputer.
Technical Report Ultracomputer Note No. 28, New York University, May, 1981.

[So 86] So, K., *et al.*
PSIMUL - A System for Parallel Simulation of the Execution of Parallel Programs.
Technical Report RC 1164 (#52414), IBM T.J. Watson Research Center, January, 1986.

[Stamos 84] Stamos, J.W.
Static Grouping of Small Objects to Enhance Performanc of a Paged Virtual Memory.
ACM Transactions on Computer Systems 2(2):155-180, May, 1984.

[Steinberg 86] Steinberg, S.A. *et al.*
The Butterfly Lisp System.
In *AAAI-86: Fifth National Conference on Artificial Intelligence*, pages 730-734. American Association for Artificial Intelligence, August, 1986.

[Stolfo 83] Stolfo, S., D. Miranker and D.E. Shaw.
Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence.
In *Proceedings of 8th IJCAI.* 1983.

[Svobodova 76a] Svobodova, L.
Computer Performance Measurement and Evaluation Methods.
Elsevier, 1976.

[Svobodova 76b] Svobodova, L. and R. Mattson.
The Role of Emulation in Performance Measurement Evaluation.
In *International Symposium on Computer Performance Modeling Measurement and Evaluation.* ACM-SIGMETRICS, 1976.

[vanBiema 86] van Biema, M., Miranker, D.P. and Stolfo, S.J.
Do-Whiles Considered Harmful in Production System Programs.
In *Proceedings of the First International Conference on Expert Database Systems.* 1986.

[Wulf 72] Wulf, W.A. and C.G. Bell.
C.mmp -- A multi-mini-processor.
In *Proceedings of the Fall Joint Computer Conference.* 1972.

[Zorn 86] Zorn, B.
Multiprocessor Garbage Collection in Lisp.
1986.
Abstract of Ph.D. thesis in progress; from PARSYM digest.