U.S. DEPARTMENT OF COMMERCE • National Bureau of Standards

NBSIR 85-3296

Institute for Computer Science and Technology

Performance Measurement Techniques for Multi-Processor Computers

John W. Roberts

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Computer Systems Engineering
Institute for Computer Sciences and Technology
Gaithersburg, MD 20899

CMRF
COMPUTER MEASUREMENT RESEARCH FACILITY
FOR HIGH PERFORMANCE PARALLEL COMPUTATION

February 1986

Partially sponsored by the

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209
ARPA Order No. 5520, July 23, 1985

NBSIR 85-3296

PERFORMANCE MEASUREMENT TECHNIQUES FOR MULTIPROCESSOR COMPUTERS

John W. Roberts

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Computer Systems Engineering
Institute for Computer Sciences and Technology
Gaithersburg, MD 20899

February 1986

Partially sponsored by the
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209
ARPA Order No. 5520, July 23, 1985

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, ***Secretary***
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, ***Director***

TABLE OF CONTENTS

PERFORMANCE MEASUREMENT TECHNIQUES FOR MULTIPROCESSOR COMPUTERS

JOHN W. ROBERTS

A wide range of possible measures for multiprocessor computers is discussed, along with the realizability of each class of measurement techniques and the applicability of the results.

Key words: Computers; Multiprocessor; Measurement; Performance.

1. INTRODUCTION AND GENERAL DESCRIPTION

1.1 Introduction

Since portions of this paper may be of interest to readers with a wide range of technical backgrounds, the first section has been included to familiarize the reader with the terminology as it is used in this paper, as well as many of the concepts that are of interest. Emphasis has been placed, in this section and throughout the paper, upon the areas that seem to be of greatest concern to those who are currently designing multiprocessor systems, and which are often new areas, unique to the new systems (or nearly so). There has therefore been some tendency to gloss over many performance areas, such as throughput and latency on input/output, that are also of concern in high-performance uniprocessor systems. It should be remembered that such areas are at least equally important to overall performance of multiprocessor systems.

1.2 Processors

The fundamental unit of any multiprocessor system is the processor. There have been many philosophical arguments on the subject, but it is usually agreed that a processor is a device which takes a data input, follows a desired sequence of instructions involving the data, and produces an output. A processing system consists of a processor plus supporting structures such as memory, input/output devices, and fixed programs, which aid the processor in interface with the user(s) and execution of user programs. In its broadest definition, a multiprocessor system is any system which allows more than one processor to operate at a time. There is also usually some degree of coordination among the processors, in the matters of timing and the use of shared system resources.

Some of the processors may be special-purpose devices designed to handle part of the load of the general-purpose processors. Among these are sophisticated memory controllers, units for general mathematical computation (i.e. floating point processors), and specialized assemblies such as array processors.

Multiprocessor systems were originally designed with the aim of eliminating the bottleneck caused by running all the computational work through one processor. The first sucessful systems had a single processor in charge, with other processors as slaves. The concept has been greatly expanded since that time, both in the designs and in the philosophy of those designs. Processors may work as equal partners in a system, on a single task or on separate tasks.

A great number of different multiprocessor designs have been developed to date, and a much larger number are theoretically possible, with such variety that there is little that they all have in common other than the definition of having more than one processor. However, most components in systems now existing or expected to exist in the near future are based on a set of options sufficiently limited in scope to permit reasonable analysis. It is therefore practical to describe current systems and those that might reasonably be designed based on current knowledge, while realizing that this does not cover the area of all possible multiprocessors.

1.3 Coupling of Processors, Sharing of Resources

As an extreme design case, a multiprocessor system can consist of a number of completely independent processing systems, doing unrelated work and joined only to save costs in power supplies, cooling, floor space, etc. Most systems, however, have at least some sharing of resources among the processors, which may include input/output devices, primary memory, and secondary memory. There is also some mechanism for communication among the processors. This communication may be strictly for low-level interactions such as resolution of contention of shared resources, or it may provide the high-level interactions needed for putting several processors to work on different parts of the same job.

1.4 Memory

Primary memory may exist in a system as a resource shared among the processors, in which case it is called shared or common memory. If each processor exercises exclusive control over a fixed segment of memory, it is called private memory. It is common for a system to have both private and shared memory.

Multiprocessor systems generally have an interconnection structure such as a common bus or a switching network. If processors must use it to access a given segment of memory, that segment is usually called global. If the memory may be accessed directly without use of the interconnection, it is termed local. The assignment of local to private memory and global to shared memory is widely, but not universally used. Some systems allocate portions of global memory for private use. Others use all or part of local memory as global address space. (This means that one processor can address a given block of memory locally, while all other processors can address it globally.) As previously implied, there are also systems with only global shared or only local private memory.

When the design of a system emphasizes global memory, there is often contention among the processors for use of the interconnection mechanism, which can make average memory access time a system bottleneck. To partly get around this problem, many systems use special local, private memory structures called caches.

In addition to the usual memory organizations, some systems use more exotic structures. Associative memory, for example, is addressed partly by content, and has been built into several systems.

1.5 Cache

A cache is a local memory designed to hold a copy of part of the contents of shared memory. Since the processor does not need to use the global access mechanism to deal with the cache, there are often significant reductions in the loading of the global access mechanism and the perceived global access time. The cache memory, being smaller than the memory it maps, is often built using faster (and more expensive) components, which can further increase its speed advantage. (It is mainly for this reason that caches are often put in single-processor systems.)

A typical cache is divided into sections called blocks, each of which can hold a section of data from main memory of a specified size. Data are normally read into the cache in units of blocks. The main memory is partitioned with fixed boundaries for possible block transfer, so two blocks of the same type never overlap (some systems may cache instructions and data separately, and an instruction block and a data block could have the same addresses). The low-order address bits for each block correspond to the address bits for the cache blocks to which they are mapped. Associated with the cache is a table for all the blocks in the cache giving the high order address bits of the shared memory block which corresponds to each cache block. Management of the cache is handled by a cache controller, which hides all the details of caching from its processor. It uses the table of high-order address bits to determine whether a given block from main memory is present in cache, and other records to determine whether the block is currently a valid copy from main memory. The successful location of a requested element in the cache is called a cache hit. If the search is unsuccessful and the controller must address main memory, it is called a cache miss. With a cache hit, memory access usually takes less time than direct access of shared memory. A cache miss and the resulting access takes longer than a direct access would. Maintaining a high hit ratio (cache hits divided by total number of accesses) is therefore a very important goal.

With local caches, there are often several valid copies of a given memory location in the system. There must therefore be an agreement on when a location in cache and main memory should be updated or marked invalid, so an invalid copy will never mistakenly be used, and a valid copy will always be available. The cache controller follows an algorithm that specifies at what times cache and shared memory should be read from and written to as the processor goes about reading and writing data and executing instructions. There are many cache control algorithms now in use, and all have strong points and weak points. Some of the more widely known algorithms are as follows:

> Write through - On write, send data to main memory and corresponding location in cache (depending on allocation rules; see below). Also load cache on read from main memory.

> Write by - On write, send data to main memory and mark corresponding location in cache bad (invalid). The cache location will be restored on the next read, as for write through.

> Write back - On write, send data only to cache, making sure the main memory location is marked bad. When part of the cache must be replaced or another processor needs the data, transfer out of the cache takes place.

For all of these approaches, it is necessary in a multiprocessor environment for each cache controller to be aware of the activities of the other controllers involving blocks it contains. If the copy being used as a reference (always in main memory except in write back systems) is changed, the copies in the caches must be updated or marked bad.

There are also the issues of when to read data into cache (allocating space for new blocks), and of cache replacement. Most systems allocate space in the cache after a cache miss on a read attempt, which causes an entire block to be loaded from main memory into cache. Some systems also allocate space when the processor writes to an address that is not in cache. (This "allocate on write" policy has not been shown to improve cache hit ratios, so many systems avoid it.)

Some cache controllers allow each block in cache to map to only a subset of all the blocks in memory. A group of such blocks collectively are able to contain any desired combination of blocks from main memory. This group is known as a set. There may be several sets, and when a new block is to be loaded into the cache, it may be written into any one of several cache blocks (each set containing one block that could be used). The block to be overwritten is chosen by the cache replacement algorithm, which may include such features as picking empty or invalid blocks first, choosing the least recently used, or a fixed pattern or random choice. The least recently used algorithm is simple to implement if there are only two candidate sets and difficult to implement if there are more than two. In use, this algorithm seems to be very good at displacing the least-needed blocks, and it can be supplemented with other techniques for even better performance.

The above description applies to most caching techniques currently in use in multiprocessor systems. It should be remembered, however, that other techniques are being used and that many more are potentially possible. The consideration of caches is important since they are a vital part of a wide class of multiprocessor systems.

1.6 Coupling of Processors and Memory

A great variety of methods are used to couple processors and nonlocal memory to allow global memory access and interprocessor communication. One of these, the bus, is an extension of the design of many uniprocessor systems. The processors and the memory "modules" are all connected to a single path designed to be used by one device at a time. A single line (for serial communications) or multiple lines (for parallel communications) may be used. In any event the bus represents a single point of interconnection. The devices on the bus take turns using it, following some type of arbitration scheme. The devices on a bus are usually close together and communicate freely; therefore, a bus system of this type is considered "closely coupled".

A bus system has the advantages of being closely coupled (if this is needed), fairly simple, generally expandable, and often comparatively economical. Its disadvantages are based mainly on the fact that everything must communicate over the bus, and as the number of devices increases, the bus rapidly becomes overloaded and delays increase, slowing down the system. A caching scheme, designed to reduce interconnection traffic, is therefore particularly important in a bus system. Some systems use several buses in parallel in an effort to increase bus throughput.

The second major approach to interconnection is the network. Networks have a far greater number of potential configurations than buses, and are used in a much wider range of designs (though the numerical majority of commercial multiprocessor systems built to date are bus-oriented.) Basically, the processors, memory, and other devices are connected by what is effectively an internal communications network, over which messages pass. The internal networks may use communications protocols developed for distributed networks, but most of those designed thus far use specialized protocols optimized for performance under the conditions found in the system. These conditions usually include short propagation delays, a limited number of devices, and low error rates, so handshaking routines, device addresses, and network error detection are usually held to a minimum.

One very popular type of switching network uses a single multipath switch, with processors connected to one side, and memory organized into independent modules connected to the other. Processors usually communicate with one another by placing messages and data in shared memory. Other common types look more like local area networks, etc., with star, ring, or mesh topologies. It is interesting to note that buses can be considered a small, specialized subset of all possible networks.

There are many other important hardware and software features that are of interest in multiprocessor design, which are discussed in more detail in the various sections of the paper.

1.7 Measurement

The measurement of performance of multiprocessor computers involves a compromise between the desired and the possible. The following discussions are intended to shed light on both of these aspects. The measurement of multiprocessor computers differs to some degree from uniprocessor machines. The critical aspect of this difference is communication among the processors. The following sets of measurements are directed toward quantifying thse aspects of a particular hardware/software architecture and implementation.

2. MEASURABLE PHENOMENA

In computer systems, there is only a limited set of useful basic parameters which can be measured. These are *TIME* (related to a time reference), *TIME INTERVAL*, *TIME DURATION*, and *FREQUENCY* (events per time period). Ratios are one extension of this basic set, and may be determined by retrospective analysis or in real-time using specialized hardware. For example, *DUTY CYCLE* is the ratio of cumulative time duration of an intermittent event to the total time interval of the measurement.

Intervals can be measured by observing the start and stop times of each event, or by assigning a hardware time measuring circuit to each event channel. It is particulary important that time be expressable in terms of number of clock cycles, since this is a more basic measure of an architecture than time alone.

In many cases the values taken for a particular type of event will vary between successive measurements. It can be useful to record the distribution of these values, especially in a manner which can be correlated with some other event.

3. DETECTION OF EVENTS

The problems associated with the detection and measurement of events fall into the two general categories of gathering and reducing the data. In gathering data, a needed item of information may be difficult to find, requiring the use of complex hardware devices or extensive processing. Once the measurements have been acquired, the sheer volume of data available can make storage and processing very difficult. One solution is to partially process the incoming data to reduce size and bandwidth requirements on the storage device.

A few examples illustrate the issues encountered in the detection and measurement of events. Some events, for instance Bus Access Requests (from each processor), and Bus Grant signals (to each processor) can usually be detected merely by observing the state of a single wire. The delay between assertion of these two signals represents the Bus Access Latency. Since these events may be very close in time, a hardware timer may have to be provided in the measurement hardware to reduce the data which must be recorded. A further reduction of the amount of data which must be recorded could be made if the distribution statistics for Bus Access Latency were obtained by measurement hardware prior to recording. Any such "winnowing" of data prior to recording naturally reduces the possibilities for later analysis, but may well be necessary in most practical situations. This points up the need for careful experiment design.

More complex events can be detected by observing the address values in the course of execution of test programs. (It would be particularly useful if assemblers and language compilers were modified to allow the programmer to specify measurement trigger conditions in the test programs themselves.)

It is felt that measurement event trigger hardware, in order to be really useful, will necessarily resemble the event trigger portion of the Transaction Analyzer [GAUD]. This device incorporates a state machine which traces the flow of events (many simultaneous signals) and detects desired measurement events.

4. BENCHMARKS

In order to perform measurements of most higher-level functions, the system must be executing one or more user programs. Programs written for use in testing processing systems, taken as a class, are known as benchmarks. There may be a large set of benchmark programs for any particular system, all tailored for specific types of measurement.

Many benchmarks are designed to provide a realistic estimate of overall system performance. This is a difficult job because the performance of a given set of user programs can be highly dependent on whether or not they take advantage of certain machine-specific features such as vector or array processors, pipelines, math coprocessors, high-speed memory, etc., and how efficiently they use these features. Benchmarks of this type, therefore, often tend to be very similar to the anticipated user programs, and sometimes actual user programs are employed. For this and other types of benchmarks, a profile of the intended workload can be very useful in developing a benchmark that will predict performance with reasonable accuracy. If the anticipated workload is very uniform in nature, development of such a benchmark can be a relatively simple matter. If, however, the system is expected to perform a wide variety of different jobs, benchmarks will be difficult to produce and will generally be poorer as predictors of performance for particular cases, since the relative effects of machine-specific features on greatly differing programs can be highly unpredictable. In cases like this the need for multiple benchmark programs to predict average performance is emphasized.

Other benchmarks are designed mainly to test certain features of machine architecture, operating system design, etc. While the properties of intended user programs may have some effect, these benchmarks are usually more concerned with ultimate performance limits of a cache, I/O controller, communications path, or other element, rather than performance under typical conditions. Therefore the test programs usually do not represent typical user programs.

In the design of benchmarks, one often wants to detect idiosyncrasies of the machine, such as certain array sizes that lead to unusually good or unusually poor performance, both because these can lead to highly misleading benchmarks and because they are something programmers should know about. For this reason it is often a good idea to develop a suite of benchmark programs, with a number of different approaches used, and with variations in certain parameters (array sizes, timing loops, number of interacting processes, etc.) as controlled variables. This approach is especially important for multiprocessor systems, since their extreme complexity can create many nonobvious pitfalls.

5. ORGANIZATION OF THIS PAPER

The part of this report describing techniques is divided into portions, each associated with some aspect of multiprocessor systems. Since many parts of a computer have features analogous to one another, and since specific features may be important to more than one general topic, there will inevitably be some overlap. Each portion includes a discussion of the aspects of computer performance which can be measured, the detailed measurement techniques required, and some important uses of these measurements.

6. ORGANIZATION OF FEATURES

6.1 Sample Entry

Measurement technique

This paragraph is used to describe one or more suggested approaches to making the measurement, and probable difficulties that will be encountered with each.

Possible uses

1. Evaluation - This paragraph discusses the use of this measurement in evaluating the performance or design of an existing computer.

2. Tuning - This paragraph discusses use of this measurement by the computer user in order to improve the performance of the computer on the application under test.

3. Future design - This paragraph discusses the application of the knowledge gained from this measurement to computer design or redesign, or to the development of future measurement equipment.

7. PROCESSOR

7.1 Time Spent Reading, Writing Data

Measurement technique

Logs of accumulated time spent reading or writing data may be kept by timers which are enabled by local bus control lines. Well-timed random or periodic sampling should also give satisfactory results. Times for particular access types (write/read, user/supervisor, etc.) should be measured using timers and detectors adapted for each type.

Possible uses

1. Evaluation - Time spent reading and writing data relative to execution time is a measure of the appropriateness of the data memory scheme to this particular processor.
- Relative frequency of data accesses as compared to total execution cycles can be used to study the fundamental properties of software.

2. Tuning - Average times or distribution of times for particular types of accesses can guide selection of data structures and placement of data, and adjustment of program algorithms.

3. Future design - Overall relative time should be used to help decide how much emphasis to place on data memory design and whether to design specialized data-handling mechanisms.
- Times for particular types of access determine their performance and allow informed selection of cache mechanisms, local memory, etc.

7.2 Time Spent Fetching, Executing, Instructions

Measurement technique

As with time for reading and writing data, most of the instruction read time measurements can be taken using state detectors that look at each processor's local bus and activate counters and timers during instruction reading. Statistical sampling could also prove useful and reduce the amount of data taken.

Possible uses

1. Evaluation - Access time for instructions relative to their execution time is a measure of the appropriateness of the instruction memory scheme to this particular processor implementation.

2. Tuning - Overall times can be observed for accessing code arranged in a variety of ways, including large loops, small loops, linear code, disjoint code, etc. The results will be partly a function of division of memory into global, local, and cache sections, and the management of transfers between them. This information can be used to help choose code structures for best performance.

3. Future design - Determination of the relative amount of time spent fetching instructions will help in

deciding to what extent work on better instruction management schemes is warranted.

7.3 Time Spent Waiting for Bus Access, Cache etc.

Measurement technique

Average time and proportion of time can be measured using counters controlled by sensing the control signals on the local processor bus. For this particular area of measurement, intervals are traditionally measured in increments of whole system clock cycles, which are all the processor may be expected to care about. Measurements should be tied in with the ratios of various types of instructions observed in the code.

Possible uses

1. Evaluation - Comparison can be made of performance of the nonlocal access devices against local memory performance.
- Proportion of time spent waiting for responses as part of overall processor active time is a measure of the effectiveness of the utilization of the processor. (Active time will be defined as time during which a processor is not halted and is working on some task.)

2. Tuning - Knowledge of average time per request makes the programmer aware of the costs of the various types of access.

3. Future design - This will likely indicate important bottlenecks.

7.4 Time Waiting for Synchronization Signals

Measurement technique

Because of the different kinds of synchronization mechanisms, techniques to detect the signals and measure the waiting times must be adapted to each particular system. Common memory systems can be observed using address recognizers for both polling and writing of key addresses (which are determined at compile or loading time). Bus-oriented systems may have monitors to look at control lines or control packets. Analogous techniques can be used for network-based systems.

Possible uses

1. Evaluation - Synchronization signals are used in a multiprocessor environment to coordinate the activities of the various processors. As an example, a given processor that is cooperating with several other processors on a user or system job may have accomplished all it is capable of doing without further interaction with the other processors. It may then send, receive, or both send and receive, synchronization signals, which may contain data and/or control information. The exact mechanism may range from writing to and polling memory locations to sending and receiving packets. If an inordinate amount of time is spent in this process, then there is something wrong with the task division, the synchronization mechanism, or both.
- The average time waiting for synchronization signals and a log of all the waiting times helps to determine how well the programming is taking advantage of the multiprocessing capabilities of the machine.

2. Tuning - In systems with fully user-controlled allocation of tasks to processors, a log of all

waiting times is needed for adjusting the allocations for better performance in programs that will be run many times. Average waiting time may be used as a measure of the necessity for further tuning. (In any event, the fact that subsequent runs with different inputs and different competing processes may have different timing makes precise tuning of this sort of system very difficult.)

3. Future design - An analysis of waiting times could lead to improved allocation schemes (presumably automatic, probably dynamic) and other automatic tuning mechanisms.

7.5 Inactivity Due to Hardware Idle Cycles

Measurement technique

Measurements of processor idle cycle activity are simplest if there is a way to access the signals to the processor, preferably "wait" or "hold" signals. Observation of other control and address lines will be required to identify the reason for the wait.

Possible uses

1. Evaluation - This is a more general case of the measurement of time spent waiting for accesses , and includes other reasons for temporary processor inactivity. For example, the memory management hardware of many multiprocessors can stop the processor for one or more clock cycles even when accessing local memory. A processor may have some fundamental request, such as a system bus grant request, which is arbitrated by system hardware and which will frequently not be granted right away. Analysis of average number of hardware idle cycles broken down by cause can help to determine whether the resources of the system are adequate for the needs of the processors.

2. Tuning - Knowledge of the numbers of hardware idle cycles (and overall completion times) typically associated with various processor activities can help when programs are being optimized for speed.

3. Future design - By observation of the number of hardware idle cycles used in various actions and the overall number in relation to total cycles, one may determine what hardware mechanisms most need improvement.

7.6 Processor Inactivity while Waiting for Work

Measurement technique

Observation of periods in which a processor is inactive because it has no work to do must be tailored to the type of multiprocessor system being measured. Some have available signal lines indicating the inactivity of each processor, while for others it is necessary to document the activity of each processor based on local bus addresses, etc. In systems using dynamic allocation of processes to processors, it may be possible to observe the length of the process queue and the number of processes being executed at any given time, hopefully without excessively perturbing operation.

Possible uses

1. Evaluation - The total amount of processor inactivity shows the extent to which the full capabilities of the multiprocessor system are not being utilized. When observed, it can be because there is not enough work to keep all the processors busy, or it can be a result of poor resource management. An analysis of a pattern of inactivity should help to determine which is the case.

2. Tuning - A general record of inactivity as a function of time shows what capacity may be available for other jobs.
- A program partitioned into tasks running concurrently on such a system could be tuned by repartitioning it for higher utilization of the processors based on a record of inactivity.

3. Future design - The record of processor inactivity gives designers an idea of the processing capability required for a variety of jobs.

8. MEMORY

8.1 Addresses Used by Various Applications

Measurement technique

Address detection can generally best be done by use of bus monitors that look at address and control lines at various locations and determine what addresses are being read from or written into. If a histogram is desired on the basis of pages in memory, a number of the least significant bits can be ignored. Similar methods can be used for systems which partition local and shared memory by address.

Possible uses

1. Evaluation - Processes typically have most or all of their code in a contiguous block in memory with a given address range. It is often possible to identify a process being executed (within a certain context) by the value of a logical or physical address. Thus an observer could determine which processes were running on which processors at any given time by looking at the memory locations addressed. For a shared memory system, if the addresses to be observed are the physical addresses, then a single monitor could observe all processes being executed.
- In shared memory systems with additional local memory for each processor, the distinction between local and shared memory for each processor is usually made by address range. Thus by making a simple histogram of the addresses found on the local bus, one could determine how much of the addressing done is to the local memory and how much is to the shared memory.
- Relative frequency of given address ranges gives an idea of the relative activity of given functions.

2. Tuning - Using a histogram of memory addresses to determine which functions take place as desired, one can adjust programs for correct operation.

3. Future design - If study shows that there are a large number of instruction/data references which are

of purely local interest (and which would therefore unnecessarily tie up a shared memory bus), then the possibility of enlarging local memory could be investigated.

8.2 Global Memory Activity, by Address

Measurement technique

In shared-bus shared memory systems, memory address accesses may be monitored from the system bus address lines. Storage is by counters, memory tables, registers, or sequential memory locations in the measuring equipment. In network-based systems, the monitoring equipment must decode the addresses in the packets or look directly at the memory modules. In any case a high data rate will make expensive monitoring equipment necessary. Research should be done to evaluate the use of cheaper sampling equipment for this measurement.

Possible uses

1. Evaluation - Global addresses can be taken to refer to physical addresses over the entire address space in some systems. The efficiency of use of the address space over the areas that contain memory or memory-mapped devices reflects one aspect of efficient memory use.

2. Tuning - A record of global addresses used during execution of a program gives evidence for unused resources that can be reallocated for greater system efficiency.

3. Future design - A collection of memory usage histograms could show whether there is a sufficient amount of primary memory for the efficient operation of all the processors. (If there is not, the resulting frequent swaps between primary and secondary (i.e. disk) memory will slow down the system.)

8.3 Relative Frequency of Reads and Writes

Measurement technique

Read and write commands are normally found as control signals on the local bus and often on the system bus, where they can be picked up by probes. (System commands are also found in control fields of packets, and must be decoded for analysis.) Other accesses that can often be picked up this way include read-modify-write cycles and certain types of address calculations. Addresses can be picked up along with the control signals. In machines with system bus arbitration, a monitor must keep track of the current owner of the bus in order to identify the processor originating the request.

Possible uses

1. Evaluation - In shared memory multiprocessor systems with local cache, the relative frequency of reads and writes is very important in determining how well the cache system will work to reduce loading on the main memory and system bus. (This is true both for the common cache techniques and for the ownership (write-back) approach as used in at least one modern system.)
- A comparison of the ratios of reads and writes along with the overall number of accesses between local bus and system or memory bus gives a direct measure of how well the nonlocal traffic reduction techniques (i.e. caching, local memory) are working.
- If there are reads and writes to memory locations used as synchronizaton mechanisms, a record of all such reads and writes (as well as the overall average) shows how well the processes are coupled and to

what extent this mechanism loads the system bus.
- Sequences of reads and writes can be arranged so values do not have to move around much on the system bus, or they can be such that data must be transferred nearly every time. The arrangement of sequences can therefore have great impact on system performance. The exact relationship between sequences and required accesses is a function of cache technique and memory organization.

2. Tuning - If a program is found to have an undesirable relative frequency of reads and writes, it can often be partially rewritten to give a more favorable ratio for better system performance.
- If the sequence of reads and writes is found to be lowering performance, rearrangement of the code may result in a better sequence.

3. Future design - Knowledge of the general pattern of reads and writes expected can help greatly in the design of caches and layout of memory. This was probably a major impetus in the development of the cache ownership mechanism.

8.4 Memory Allocation to Code, Data etc.

Measurement technique

A linkage map, generated by the linker-loader program, is probably the best means to obtain a detailed memory allocation map. Some linkers may not identify the usage as well as desired, i.e. code, data, local variable, global variable, synchronizing variable, semaphore, etc. In this case, one can obtain useful, but not complete, data by an analysis of memory locations accessed during program execution.

Possible uses

1. Evaluation - If memory size is a constraint in a particular system, allocation of program and data space gives a measure of the space efficiency of the compiler, which can be important. Also, one can determine how much of the code and private data will fit in private memory.

- This map should show any obvious inefficiencies in memory allocation.

- For systems with cache, the amount of address space allocated to data, along with the caching technique and patterns of data access and execution (if they can be determined ahead of time), allow one to estimate how well the caching technique will work in any particular instance.

2. Tuning - Knowing the memory allocation of a program and the memory structure of the system on which it is to be run, it is often possible (when there is sufficient interest) to modify the program so it will run more efficiently.

3. Future design - If many programs turn out to inherently require certain memory structures for optimum performance (as determined partly from studies of memory allocation), then it would be a good idea to consider implementing these memory structures in future systems.

8.5 Effect of Context Switches on Above

Measurement technique

Context switches usually result in a change in memory allocation. They are often associated with interrupts. If that is always the case, a monitor could determine when these interrupts occur. At the same time, other monitors could make logs of the parameters of interest. Other promising methods of detecting context switches include looking at the process queue (if any) and looking for certain particular shifts in addresses.

Possible uses

1. Evaluation - Since exchanging of active processes is a normal activity in nearly all computer systems, and especially in multiprocessor systems, the effect of context switching is a significant area of concern. When swapping occurs, the old process usually has data and sometimes code that must be moved out of the current memory and stored elsewhere. New instructions and data must then be moved in. This process may take place all at once, or gradually as new locations are needed. One would expect to observe the greatest effects of swapping in cache and local memory. In any system with fairly frequent context switches, it is important to observe the processing overhead associated with the resultant memory transfers.
- The main change in memory to be expected during a context switch is in the pattern of reads and writes. It is important to observe this change, and its effect on cache hit ratios, etc.

2. Tuning - With an increasing frequency of context switches, it is increasingly important to minimize the harmful effects of such switches on system performance. There often appears to be a tradeoff between processes designed to execute efficiently, and processes that can be switched efficiently (code size, local data structures, etc). Knowledge of the effects of context switching allows intelligent choice of switching patterns and code and data structure design.

8.6 Storage Transfer Algorithms

It is useful to look at the algorithms used to load data from secondary storage into main memory. The way in which this is done partly determines the way in which the memory management unit operates, which in turn influences the patterns of invalidation and replacement in the cache controller. The loading algorithm therefore has a great influence on system performance.

8.7 Reservation of Address Ranges

These measures must be adapted to the cache control scheme used in any particular system.

9. CACHES AND LOCAL MEMORY

Caches make use of a variety of techniques in an attempt to allow access to global memory values with only slightly more delay (on the average) than local memory accesses. They are described in some detail in the introductory section of this paper.

In some systems, some local memory is used along with or in place of a local cache. If instructions or data of purely local interest are placed in this local memory, a great deal of interprocessor or shared memory traffic can be avoided.

9.1 Overall Hit Ratio

Measurement technique

The most straightforward technique to measure read hit ratio is to count memory access requests on the bus to the cache, the local memory, and on the bus to main memory, originating from the processor of interest. In most systems with local cache, a cache miss upon reading will lead to a block transfer into the cache. The result of a miss on write is heavily dependent on the caching technique, and there are a wide range of such techniques currently in use. It may be possible to tap into signals coming from the cache that reveal, by state or timing, whether the cache access resulted in a hit or a miss.

Possible uses

1. Evaluation - The overall hit ratio is one of the most important measures of a caching system, and, with information on typical access latencies associated with hits and misses and the costs of block data transfers, serves as a reliable predictor of cache performance.
Local memory hit ratio is a measure of the success in placing instructions and local data in this memory.

2. Tuning - Cache hit ratios can be expected to vary from program to program and under different conditions. Since the hit ratio is a major determinant of performance, it is often worthwhile to adjust a lengthy or often-used program for a high hit ratio.

3. Future design - Looking at the various subdivisions of hit ratios and time penalties for the set of possible actions, one one can determine the value of caching and local memory for a particular system and work on ways to improve performance.

9.2 Instruction and Data Hit Ratios

Measurement technique

Most modern processors have control signals which tell what kind of access is being attempted, i.e. instructions or data. These events should be counted in order to contrast the behavior of instruction and data accesses, at the same time monitoring the overall behavior of the cache and local memory. If these signals cannot be accessed, more difficult techniques involving address recognition may be necessary.

Possible uses

1. Evaluation - In most systems there are many differences in the treatment of instructions and data. Since the principle of caching and local memory is to take advantage of certain memory access patterns, it is clearly difficult to design one cache that is optimized for both instructions and data. (This problem is separate from that of instructions and data interfering with one another.) It would therefore be useful to know the level of performance for instructions and data.
- If the system in question has separate caches or local memories for instructions and data, they should

naturally be evaluated as distinct entities.

2. Tuning - For both instructions and data, cache performance can be significantly reduced in certain programs by "unfortunate coincidences". With instructions, hit ratios are reduced significantly if execution jumps around a lot, especially if the jumping covers areas that cannot all be in cache simultaneously. Hit ratios are reduced drastically if a process frequently calls a subroutine and the addresses happen to be such that the subroutine replaces in cache the code that called it. Similarly, for data structures it is possible to set up a structure with a certain organization and size, and access its elements in a certain sequence, so that cache blocks constantly replace one another and the hit ratio drops. These phenomena will probably remain fairly constant over repeated runs of the program. Therefore it should be fairly straightforward and very profitable to look for unexpectedly low hit ratios, and tune instructions and data by moving parts around, changing sequences, etc.

3. Future design - Looking at current cache performance for instructions and data can help one to decide on the usefulness of caches in general and whether future systems should have caching of:

- instructions and data together.
- instructions only.
- data only.
- instructions and data separately.

- If the above interference problems occur with many different programs and in a variety of situations, a record of the problems can help in the design of new caches and new caching techniques to get around them. In addition to the above, possible changes include larger caches, more segments, different segment sizes, different replacement algorithms, etc.
- Local memory can be evaluated in a like manner.

9.3 Relative Frequency of Reads and Writes

See same topic under MEMORY.

9.4 Distributions of Access Latencies

Measurement technique

Since processors are concerned with the outside world, mainly in the discrete intervals of clock cycles, it seems most sensible that access latencies be described in terms of clock cycles. The same sort of equipment that identifies reads and writes can pick up the starting points of access requests, then interval timers can keep track of the delays, and report to the monitoring equipment. Accesses must be identified as being satisfied by the cache, or local memory, if present. Delays can be compared to those of ideal memory, or it may be desirable to directly observe wait cycles. A complicating factor is the delay associated with memory management.

Possible uses

1. Evaluation - One of the advantages of a cache system in memory, and the main reason caches are often used in single-processor mainframes, is the fact that access latency (at least for reads) is generally shorter from cache through the cache controller than from main memory without any cache. A similar advantage accrues from close local memory. (In multiprocessor systems, the reduction of traffic on the interconnection system is also very important.) However, since normal cache hit ratios are never 100%, the performance measurement of a cache must include access latencies incurred when there is a cache

miss. These delays are usually greater than the delays of non-cache accesses, since it takes a certain amount of time to determine that the value is not in the cache. (A non-cache delay would be difficult to determine in a multiprocessor system when there is contention with other processors for system resources.) In addition to these factors is that of block size. In most caching schemes, a miss is followed by transfer of a block of data from main memory to cache. This block may be the size of a normal read transfer or it may be much larger, and the complex effects on total overhead must be fitted in. The most reasonable approach appears to be to measure average latency, and average latencies associated with hits and misses, then combine these observations.

2. Tuning - In order to realize the true implications of cache hit ratios, one must know the access latencies involved. With this knowledge, one can determine to what extent a program should be modified to enhance the hit ratio.

3. Future design - For both hits and misses, latency is an undesirable factor that designers try to minimize. With information on latencies and their sources, one can look for new ways to arrange timing and control sequencies that minimize the penalties of having a cache. As an example, some of the new microprocessors have on-chip cache, so a clock cycle for address transfer is not necessarily needed. Similar techniques can be applied to mainframe multiprocessors.
- Information on average latencies and their distributions may be used to choose tradeoffs between minimizing overall latencies and minimizing the hit latency and maximizing the hit ratio.

9.5 Cache Invalidations Due to Accesses by Other Processors

Measurement technique

In appropriately equipped systems, one can hook into the bus monitor to detect invalidation signals directly. As for other tests, it may be difficult based on system records to determine whether they are "true" invalidations. To obtain this information without perturbing the system, it may be necessary to use some kind of cache controller emulator to run in parallel with the cache system so the internal signals can be observed. Overall, simulation may turn out to be the simplest approach.

Possible uses

1. Evaluation - If the system is designed and programmed correctly, there appear to be three situations in which true invalidations (data previously valid) could take place:

1) The location represents an interprocessor synchronization variable, which, it has been decided, can safely be kept in cachable memory. If this is the case, then the invalidation is part of the normal synchronization process, and may in fact be a signal for which the processor has been waiting. Therefore, there is no untoward loss of efficiency.

2) There is another processor working concurrently with the same data. In this case, the invalidation may well be a hindrance to the operation of this processor. It is therefore important to measure the frequency of invalidations. (This analysis ignores the possibility of self-modifying code, which has fallen into disfavor, and is currently almost never used. The situation is similar, but if systems using it ever come into existence, it should be considered separately.)

3) Another processor is now working with data which this processor was working with and has now abandoned, but which has not yet been thrown out of the cache. Since it is very difficult to determine when this is the case if there has been no context switch, and since context switches in systems with memory management typically throw everything out of the cache, it may be that this possibility,having minimal effect, is best ignored.

A complicating factor for this type of measurement is the fact that many systems may have no logical means to determine whether the "bad" flag or its equivalent was previously set before setting it. This would then have to be implemented in external hardware (if possible), which may may make it difficult to determine whether a "true" invalidation is taking place. Another processor repeatedly writing to a given location might cause multiple invalidations of a location already marked invalid. Obviously, this would have less impact on performance than true invalidations.
- The system may not be designed and programmed correctly, in which case the invalidations would represent faults. Knowing what is being done at any given time and looking for unexpected invalidations, one can detect at least some of these faults.

2. Tuning - If, in a multiprocessor system, several closely- coupled processes are simultaneously working on the same data, frequent true invalidations can lead to high average access latency. If this is found to be the case, one may be able to change the programming so the processes do not interfere with one another as much. As an example, several processes simultaneously starting to work on an array might be able to start at different places in the array.
- If two processes are to run simultaneously, reading from and writing to adjacent fields in memory, and if the border between the two fields happens not to fall on a block boundary, then it is likely that the two processes will interfere with one another much more than is reasonable, causing many normally unnecessary invalidations. Observations can be made to detect this type of situation, and memory allocations can be adjusted, resulting in slightly less efficient use of memory but much better performance.
- If it is determined that some of the invalidations are caused by faults, hardware or software fixes can be initiated to cure the problem.

3. Future design - If the above parameters prove difficult to measure,it may be an indication that future cache controllers should be designed so the needed information is easier to access from the outside than in current systems.
- It may be possible to design new systems in which cache invalidations do not occur as often.
- There are several advantages to making invalidations apply to less than a whole block. This would be useful if the other values in the block are to be accessed locally, and would be especially helpful with large block sizes.
- Measurements like this may show that cache blocks should be relatively small in order to minimize the effects of invalidations, and can help to establish an optimum size for best cache performance.

9.6 Cache Invalidations Caused by Local Processor Access

Measurement technique

The same techniques could be used as for observation of invalidations caused by other processors. One could identify self-invalidations by looking for the originator of each invalidating memory address, but it is probably preferable to run the process in question by itself, with all other processors stopped, in order to directly observe all such events.

Possible uses

1. Evaluation - Typically, this situation arises when a processor writes to a location which is currently represented in cache. In write-by and related caching schemes, this causes the value to be written to main memory and the location in cache to be marked bad. In current systems, the entire cache block will be marked bad. The effects, therefore, range beyond just one memory location. Performance can suffer if the block must be repeatedly loaded into cache. This is particularly true if the block size is greater than the width of the data bus. If the blocks are very large, performance can become very poor.

2. Tuning - If there are a number of addresses in memory which are to be read from and written to, and if analysis shows that they are likely to be located in the same cache block, it would be helpful to orchestrate the reads and writes as much as possible in order to minimize the number of reloads of the cache required. Running the system with just one processor active and observing the number of cache invalidations, one could see how much progress had been made.

3. Future design - A number of cache control schemes that do not necessarily invalidate on write have already been developed. It seems reasonable that future designs should avoid the disadvantages of the write-by scheme, in addition to the possibility of having less wasteful invalidation.
- Using a technique such as the ownership system, a processor may temporarily become the owner of the only valid copy of a memory location. Until it gives it up, the processor may read from or write to the the location as many times as it wants without disturbing the rest of the system. A technique such as this could lead to significant reductions in invalidation.

9.7 Replacements Due to Local Processor Accesses

Measurement technique

The means to be used for detecting these events depends on the caching technique used by the system. In a typical system, a cache miss on read or instruction fetch triggers a block replacement. With controls on such effects as write replacements and interference from other processors, this should serve to record all such events. The replacements caused by moving items into cache for the first time as a new process starts up will probably not be considered a legitimate part of this measurement. In most modern systems, a new process causes a context switch and a complete invalidation of the cache. Therefore, a count of all the blocks accessed by a process may be made and that number subtracted from the total number of replacements.

Possible uses

1. Evaluation - If the placement of data in memory or the sequence of accesses or the relationship between instructions and data (in systems that cache them together) are such that blocks with the same intermediate bits are repeatedly accessed in an interleaved fashion with respect to time and there are more such blocks than there are sets in the cache, then there will be frequent cache replacements. This will result in an unreasonable loss of cache efficiency.

2. Tuning - If the situation described above arises, with blocks used by a process frequently replacing one another in cache, then system performance will be greatly reduced. The frequency of cache replacement on read associated with a single process can be used to detect this condition, and make it possible to move things around in memory (or change the code) so memory blocks accessed close to one another in time will map to different blocks in cache.

3. Future design - An analysis of the behavior of the cache in this respect can tell how well the cache is servicing the memory needs of its processor and whether improved designs are needed for the future. Among changes that reduce read replacements are increasing the number of sets in cache, and caching

instructions and data separately.

9.8 Amount of Cached Data Never Accessed

Measurement technique

There are not any immediately apparent ways to measure this parameter. Possibly something on the order of a cache emulator could be hooked up in parallel with the cache and could record all memory accesses for each element. For predictable programs, this might be a good application for a simple simulation.

Possible uses

1. Evaluation - When anything is loaded into cache from memory, a whole block of cache is loaded in a single (though not necessarily atomic) operation. If the block size is as small as the element with which the processor is concerned, then everything in the block is automatically represented in the single access. If the block size is larger, then the other data have been cached based on the statistical observations that if a given address is accessed, nearby addresses are often accessed within a short period of time, and that their prior presence in cache may considerably speed access. In fact, many of the nearby addresses are not accessed during the time they remain valid in cache. If the block size is larger than the size of the system bus, so that multiple accesses are required to load the block, these unaccessed data represent wasted effort and time expended by the cache controller and system memory bus, which can slow down the system.
- Both instructions and data are often accessed in a largely sequential manner. When this is not true, only the one element which caused the block to be cached may be accessed. If block sizes are large, this can represent a considerable amount of wastage.

2. Tuning - As previously stated, placing the beginning of a sequentially accessed chunk of data partway through a block can result in inefficient use of the cache. If this is found to be a major problem, it can be useful to rearrange the memory allocation and even waste some memory space in order to place such groups of elements at the beginnings of memory blocks. Other problems can also be found and corrected.

3. Future design - This measurement could be very useful in finding an upper boundary for block sizes, since larger blocks tend to increase the problems of unused data in cache.

9.9 Frequency of Cache Invalidations Caused by Task Changes

Measurement technique

There are sometimes signal lines from the processor or the MMU that tell when there has been a task change or when there has been a change in the translation table. Presumably the entire cache contents are invalidated at this time.

Possible uses

1. Evaluation - When a processor stops execution of one task or process and starts execution of another, it is usually assumed that it is going to work in different areas of memory from those that have been used up to now, at least for instructions. In systems with memory management, a task change generally brings about a partial or total change in the memory management unit, which affects the logical to physical memory address translation table. When this table (which may be stored in main memory in systems with a cached MMU) is changed, then the values in cache may no longer be the same as those in the corresponding memory locations (new values may have been loaded from secondary storage, etc.) and the cache may have to be invalidated. On change of task it is customary to invalidate the entire cache and start loading it again as needed, even if some of the same data are used by the new task. Since memory access can be expected to be slow for a while after an invalidation, it is important to observe the pattern of invalidations to study their impact on memory performance.
- Sometimes (as in a memory-mapped system) a given task will repeatedly load different blocks of data from secondary storage (i.e., disk) into a fixed buffer in main memory. When this happens, the mapping between logical addresses and physical addresses will be changed, and some or all of the cache may have to be invalidated. Some systems, in order to minimize the complexity of the cache controller, may invalidate the entire cache, even if only a small amount needs to be invalidated. The frequency and efficiency with which such invalidations take place are therefore of great interest.

2. Tuning - The frequency and magnitude of cache invalidations caused by task changes is a good measure of the costs associated with such changes, and determines the importance of measures to minimize the frequency of task changes. This minimization could be done by means such as dividing jobs into larger tasks, and merging related tasks into a single task.

3. Future design - Given these observations, there may be an incentive to develop cache controllers that are more efficient in their invalidations, and do not throw out blocks that are still valid. It is possible, however, that the overhead associated with a more informed controller would actually slow the cache down, so design decisions should be based on actual observations.

9.10 Frequency of Cache Misses Caused by Earlier Invalidations

Measurement technique

To detect all misses caused by earlier invalidations, one would have to keep track of all addresses that have ever been invalidated in cache and see if they are ever accessed again, which does not seem practical. If the scope of the analysis is limited to addresses that are currently represented in cache but are marked bad (which seems like a reasonable approach), then it might be possible in some systems to access internal signals in the cache controller or to keep track of cache activity using a cache emulator to detect these events.

Possible uses

1. Evaluation - These events take place when there is a cache miss on an address that has previously had a valid copy in cache, but which has been marked bad by the cache controller either after a signal from the system bus monitor, or as a result of a change in the memory management unit. It is a measure of the true penalty incurred from invalidations.

3. Future design - There is some thought that many systems extend the invalidation algorithm to invalidate cache locations beyond the minimum required. This measurement could indicate the degree of

need for systems that limit the scope of invalidations.

9.11 Frequency of Cache Misses with Invalidated, but Correct, Data

Measurement technique

There does not appear to be any practical method to measure this parameter.

Possible uses

1. Evaluation - This is a subset of the previous parameter, and refers generally to cases in which the MMU address translation table is changed, resulting in a cache invalidation, after which the MMU table and the memory are restored to their original values, during which time no new values have been loaded into the cache location, which means that the cache location contains valid data but is marked bad. This measure is an indicator of the nonoptimum character of a cache controller and its invalidation algorithm.

3. Future design - This parameter can indicate whether there is a need for a cache controller that keeps track of all invalidated locations to see if they become valid again. The implicit complexity of such a technique seems to imply that there is no practical use for such a design.

9.12 Frequency of Cache Misses Caused by Data Replacing Instructions and vv.

Measurement technique

Except where they are mixed in individual blocks, it should be practical to keep track of which blocks are associated with instructions and which with data. If there is a miss in an "instruction block" on a data access or vice versa, then the replacement takes place and interference has occurred. "Data or instruction" signals may be available directly from the processor. Cases which result from context switches and invalidation should not be included in the measurement. Such cases might be detected by determining the values of the "bad" bits in the cache control registers.

Possible uses

1. Evaluation - It may be common, in a typical system with instructions and data cached together, for a set of instructions and the data with which they are working to be placed in memory so that they map to the same cache block, thereby leading to a risk of cache interference. This problem is particularly severe in systems with only one set per cache. In many systems, the interference of instructions and data could make cache performance very poor, so a record of the frequency of these replacements would be an extremely useful measure.

2. Tuning - In a system which is susceptible to this problem, it should be possible to detect the interference in the cache and to try to relocate instructions and data in the address space so the problem does not occur or occurs with less severity.

3. Future design - Design of compilers and loaders that automatically check for this problem could be very helpful. Improved cache systems, including those with better replacement algorithms and those which cache instructions and data separately, could largely alleviate the problem. Continued measure-

ment could serve as a guide for design.

9.13 Cache Misses Due to a Poor Replacement Algorithm

Measurement technique

It may be difficult to identify which replacements are the result of "bad" algorithms, and which replacements subsequently lead to misses. Some useful results could be obtained by comparing frequency and choice of replacements for various algorithms. One could also make a log of replacements and cache misses and analyze it later to detect these events.

Possible uses

1. Evaluation - As stated in the introductory section of the paper, having two or more sets in cache generally reduces the problems of invalidation and cache replacement and tends to improve cache performance. A tradeoff for having a large number of sets is the difficulty in implementing an efficient replacement algorithm (and associated hardware) which best takes advantage of the multiple sets. As a general rule a "least recently used" algorithm works very well, though perhaps one that also keeps track of called and calling processes would be better. The LRU algorithm is usually easy to implement in caches with two sets, since a single bit per pair of blocks can record which was least recently used. With more than two sets, however, the LRU algorithm can become very difficult to implement, and substitute replacement algorithms such as "random choice" or "informed random choice" may be selected for implementation. These can perform in a suboptimum manner, not taking full advantage of the number of cache sets. While it is difficult to say exactly what is optimum, perhaps the LRU algorithm could be used as a reference with which to compare other algorithms, with comparison based on the relative number of cache block replacements (and subsequent misses).

2. Tuning - If cache replacement causes a lot of trouble, one could try to arrange the code so that replacements occur less frequently. Methods include using "larger" tasks and a reduced amount of jumping around in memory.

3. Future design - The frequency of replacements can show whether there is a need for an improved algorithm.
- If a good algorithm is going to be very difficult to implement, one could increase the block size to help reduce the frequency of repacements.
- If it has not already been done, it could be very helpful to divide the cache into sets for instructions and sets for data. These can be distinguished by the cache controller using signals from the processor, and dividing them up cuts the replacement algorithm problem in two. For instance, one could implement two instruction sets and two data sets with just a simple two-set LRU algorithm. Provisions may have to be made for the possibility of self-modifying code (see previous note).

9.14 Performance in Worst-Case Situations

Measurement technique

Some current (and probably future) systems have the ability to turn off use of cache if a high miss ratio on a given program causes performance to drop below that of a comparable non-caching system. If caching can be voluntarily turned off and the total memory access time multiplied by some factor to allow for the penalties of a cache miss, the resultant figures might give a reasonable estimate of worst-case performance.

A better method if it can be implemented is to force all the validity flags in the cache to the "bad" state, which will force a true cache miss for each access. These flags may be implemented in a discrete cache control chip, which may have a means to accomplish this through hardware, or software techniques (with accompanying distortion of the results) may be necessary. It may not be possible to perform this experiment on some systems. If all else fails, a simulation can be set up and run using worst-case parameters for cache, shared memory, etc. (with slight variations in the parameters to detect possible "resonances").

Possible uses

1. Evaluation - In most shared memory systems, access to the shared memory is probabilistic in nature. With the addition of cache, time to access any particular element is even more uncertain. While the average access time may be very good, the worst-case time is usually greater than the non-caching worst-case time and may be very poor. If the processor is working on a critical real-time application such as control of a robot or an industrial process, worst-case access times may cause occasional failures. The worst-case performance of such systems may therefore place a limit on the types of jobs that can be handled. (This is true of all systems, but the problem is enhanced by the nature of cache memory.)

2. Tuning - If a given task can not tolerate the worst-case conditions of a cache, placing it in non-cached global memory (or better, local memory) may reduce worst-case ratings to a satisfactory level.

3. Future design - For systems which are expected to run time-critical jobs, hardware inputs to force worst-case conditions would help in making sure ahead of time that the jobs could safely be performed.
- Designers may be able in many cases to produce systems in which certain previously observed conditions are guaranteed never to occur, thereby lessening the severity of the worst-case conditions.

10. SWITCHING NETWORKS

A switching network is an interconnection structure that allows a number of devices that are connected to the network to communicate among themselves. There is a wide range in such networks in degree of interconnectivity. Some networks allow only one pair of devices to be connected at a time, while others allow several connections to exist at once (though perhaps not with complete generality). An ideal generalized network would allow all elements to communicate simultaneously in any pattern, but would be very expensive, physically large, and probably slow if there were more than a few devices connected. (Networks with many interconnections tend to use lower data rates to reduce the individual cost of the many signal paths required.) A simple, inexpensive network, on the other hand, would feature a minimal number and flexibility of paths, and could have excessive blocking under heavy loads. For any network system, there is therefore a tradeoff between cost and performance.

10.1 Traffic Relative to Bandwidth for Each Path

Measurement technique

Maximum network bandwidth for each path can be determined by analysis of the physical communications medium. It would normally be considered to be the raw number of bit per second which could be transmitted, ignoring the limitations of transmission protocol. If transmissions over the network are of fixed duration, one need only know the fixed duration and then simply count the number of transmissions over each path during a measured time interval to determine traffic. If the transmissions are variable in duration, it is necessary to count each bit transmitted. If the bit transmission rate within each transmission is constant, as is often the case, a bit count may be made by measuring the accumulated transmission time.

Possible uses

1. Evaluation - The capabilities of the processors, the capacity of the network, and the demands of the programs the system generally runs are not always well matched. This parameter can show how well the capabilities of the switching network are being utilized. It will identify paths which are performance bottlenecks.
- Some of the paths in the network may have unusually low traffic. This is generally due to the nature of the programs being run (inherent or through a programming error), or from some feature inherent in the design of the network. Low-throughput paths can be a result of excessive blocking of other resources or of poor allocation algorithms. This measurement can therefore help in the detection of potential problems.

2. Tuning - If some paths are carrying too much traffic to operate efficiently while others are operating at well below capacity, it may be possible to redesign the tasks or reallocate them among the processors, to have more even loading of the network.
- If certain paths have weaknesses inherent in the design of the network, programming around them could be an effective solution.

3. Future design - If paths are too heavily loaded, it may become necessary to design improved, higher-capacity networks to meet the performance demands of future processors. This may involve increased connectivity, or higher data rates.
- If path utilization is very low, cheaper or fewer paths would suffice.
- It may be practical to correct design features such as task allocation algorithms that result in poor performance or overloading for some paths.

10.2 Network Saturation vs. Time

Measurement technique

One method to determine the point of saturation is to introduce artificial traffic requests at an increasing rate until the actual traffic on the the various paths shows little or no increase. Several different types and patterns of traffic should be used to reduce biases associated with particular traffic types. The level of traffic during actual system operation can then be observed to determine whether it is near saturation.

Possible uses

1. Evaluation - For general switching networks, there is a practical traffic limit that is below the roughly calculated theoretical maximum. It is a function of transmission length, routing, multiplexing (if any), bit rate, arbitration, handshaking, propagation delays, intermediate node delays, queueing, synchronization of the processors, error detection, collision avoidance, etc. Saturation is the point at which an increasing rate of traffic requests leads to only a slight increase or even a decrease in the useful traffic. It is important to know this point because it is the traffic limitation first encountered and is therefore the deciding one.
- Once the saturation point is known, it becomes easier to observe a running system and determine how close it is to the saturation point. The exact level of network saturation can not be precisely defined, and may vary with changes in the above factors plus patterns of access requests. In most systems, however, a determination of the degree of saturation will generally be accurate enough to be useful.

2. Tuning - When the network is carrying near-saturation traffic loads, there is generally a degradation or risk of degradation in performance. It would therefore be desirable in such cases to reduce the traffic by changing the timing of requests, etc.

3. Future design - If the saturation point of the network is too low to provide adequate service to the processors, the bus should be redesigned (or replaced with some other form of interconnection such as a network). Ways to raise the saturation point of the network include increasing the connectivity, avoiding multiplexing of instructions and data, increasing the data rate, making handshaking and arbitration more straightforward, and tuning hardware access opportunities to propagation delays.

10.3 Delays Due to Blocking, for Each Path

Measurement technique

The number and duration (and distribution) of delays due to blocking must be measured for each path. There may be hardware signals at the switching centers indicating that there is an incoming request while the switch is already dedicated. Signals can also be derived from the processors and memory controllers indicating that an attempted access has been delayed. An example of this would be a remote access request followed by a number of wait cycles. The number of wait cycles over the normally expected minimum would give the time delay.

Possible uses

1. Evaluation - Blocking in systems with switching networks for interconnection may be divided into two types of events:
1) The device to which the switched signal was directed is busy, and does not accept the signal. This sort of blocking is also found in many other sorts of systems without switching networks. The problem can be less severe if there are buffers to queue inputs to the devices connected to the network.
2) There is a "traffic jam" at some node in the network, in which the total instantaneous demand for service at one element in the network is greater than its capabilities. While one signal is being handled, the others are blocked. This leads to slowdowns and inefficiencies in the processing elements connected to the network. (Again, queueing of requests could in many cases help with short-term problems, but would increase the cost and complexity of the network.) With all of these factors to consider, this parameter is a very important one to measure.
- The distribution of lengths of delays due to blocking shows the time penalty associated with blocking and also the incidence of multiple blocking.

2. Tuning - Excessive blocking would be one of the weaknesses described for the previous parameter. If network architecture and routing are modifiable in software, this problem could be somewhat alleviated.

3. Future design - The degree of blocking encountered in the system serves as a guide for a choice in the tradeoff between minimizing the complexity and expense of the switched network and maximizing the performance.
- One modification that could reduce the effects of blocking in return for an easily-calculated expense is the placement of queueing buffers at the inputs to the switching centers. The success of this method is dependent on communications being in the general form of packets, and on the statistics of the packets and the communication medium used.

10.4 Intermittent Excessive Blocking

Measurement technique

Measurement techniques should be the same as for the overall blocking parameter, with special interest in blocking at particular nodes and timestamps for every event. Correlation with program execution may also prove important.

Possible uses

1. Evaluation - If there is a relatively short time interval during which there is a great deal of blocking, this signifies either that a feature of the network design has caused a "logjam" of messages, or that the system has been programmed in such a way as to overload the design capacity of the network at certain points in time.

2. Tuning - It may be possible to arrange the timing of the network accesses so that they do not occur simultaneously and thus jam the network. If such events are not predictable ahead of time, it may be desirable to implement something like standard communications network flow control over the switching network.

3. Future design - Same comments as for the overall network traffic parameter.

10.5 Percentage of Single- and Multi-hop Traffic

Measurement technique

No uniform method immediately suggests itself. There may be certain links that carry only single-hop traffic and others that carry only multiple-hop, and probes could monitor both of these. Links which carry both types of traffic could be subjected to timestamp analysis based upon traffic on the links to which they are connected. Switched packets usually contain routing information that could be decoded. The processors or the network interfaces may be able to provide information on routing. If the needed signals are absent, hardware modifications must be considered.

Possible uses

1. Evaluation - Single-hop traffic is that for which the sender and the receiver are just one link apart from one another in the network. Multi-hop traffic is that for which several links are traversed, with intermediate switching nodes. Since many connections have multiple routing options, there are often both single-hop and multiple-hop paths between two nodes. Other things being equal, single-hop paths are generally the more efficient in terms of speed and traffic on the net. Most optimizing routers would therefore try to select the routes with the minimum number of links, to minimize propagation delay and loading on links and switching nodes. (This consideration is more universally applicable for loosely-coupled multiprocessor systems than for generalized telecomunications networks, since differences in propagation times are essentially small.)
- If most paths that are set up have a small number of hops, then the hardware and software can work well together for optimized routing. If there are many paths with multiple hops, then there could be some identifiable factor causing apparently inefficient routing. Among the possibilities:
1) The network is handling heavy traffic loads, and the short paths are already saturated, so long alternate paths must be used.
2) A group of processors connected to the network are working together on an operation requiring intensive interaction,and are widely scattered through the network. This may very well jam the entire network, and it therefore reflects poor allocation of tasks or capabilities among the processors (or inappropriate network design if the allocation chosen is necessary).
3) Poor routing choices are being made, and the optimum paths are not being selected. Alternatively, the network has been designed so that multi-hop paths are the shortest paths in most cases.

2. Tuning - If network loading can be reduced, the short paths should be more available for use.
- When a group of processors in a multiprocessor system with a switching network will require intensive interaction, the tasks should be allocated so the processors are located very close to one another in the network.

3. Future design - If certain patterns of processor interaction often occur, switching networks in future systems should be arranged so that the paths for the often-occurring interactions are optimized.
- Certain network architectures are more likely to exhibit multi-hop traffic and are more vulnerable to its effects than others. Observation of this traffic should lead to the selection of less susceptible architectures.

10.6 Number of Active Paths vs. Time

Measurement technique

If there is no accessible central device controlling the network, active paths are probably best observed by looking at the processors or their interfaces with the network. Path occupancy, as well as time stamps (or program progress), must be recorded. One way to determine the maximum practical number of active paths is to progressively increase the traffic demand on the network and observe the point at which the increase in the number of active paths slows or stops.

Possible uses

1. Evaluation - This gives a measure of how heavily the processors are using the switching network. It should be noted that since some paths interfere with one another in most switching networks, at a given level of usage below the theoretical maximum, the network may effectively block formation of any new paths.

2. Tuning - Knowledge of wide variations in the number of active paths over time could be used by programmers seeking to smooth out usage over time for less blocking and greater total throughput.

3. Future design - If the number of active paths is often near the maximum, the designer should provide for more links or high data rate links for less blocking.

11. BUSES

As stated in the introduction, bus architectures are a specialized subset of all switching network architectures, and are of particular interest because they are used in many systems. General switching networks and bus architectures can perform approximately the same functions. All the intrasystem traffic in a bus system is routed through a single path or small group of paths instead of the many paths other switched network systems may have. While it may seem on first glance that this would be a severe bottleneck, in practice the bus usually has a much higher bandwidth (and is more expensive) than any particular path in a multipath switching network. The superiority of one design or the other would have to be decided on a case-by-case basis.

11.1 Utilization Compared to Bandwidth

Measurement technique

Bus utilization, overall and per processor, can be determined by monitoring the overall traffic on the bus and comparing it to the calculated theoretical maximum. To determine the source of any given transmission, it may be sufficient to monitor the operation of the bus monitor (if there is one), or it may be necessary to observe the bus interfaces of the individual processors.

Possible uses

1. Evaluation - The uses for this measurement for evaluation are comparable to those for switching network systems. A fundamental purpose of this measurement is to see what use is being made of the capabilities of the bus, and whether the full capacity of the bus is necessary or adequate to serve the needs of the processors. Even with local memory, local registers, and caching, the bus bandwidth is usually the main parameter limiting the number of processors that may operate in a system. With this measurement, it can be determined whether the current number of processors is close to the limit with the types of jobs currently being executed.
- In most multiprocessor bus architectures, some arbitration scheme is used in which there is effectively a hierarchy of processor access priorities at any given instant. This hierarchy may remain fixed, or may be dynamically shuffled to give all the processors a reasonable chance for bus access. A combination of fluid and fixed priorities may be used. (In a VMEbus system, for example, the processors may be divided into up to four groups, with rotating priority for the groups but with a fixed hierarchy within each group.) Sometimes, for a given priority management scheme and set of jobs, a number of the processors have disproportionately poor access opportunity, and important jobs may be delayed or blocked. Low bus utilization by a processor, coupled with numerous access requests, may indicate this condition.

2. Tuning - If this is one of the systems in which processors may readily be added and removed, the number of processors can be adjusted for the best operation in the context of bus capacity and the job

load.
- If some of the processors are blocked most of the time, one should try to reprogram the priority management for more equitable access opportunities. If this is not practical, the most critical jobs can be assigned to the processors with the best bus access for improved system performance. If no such approach is possible, some processors could be removed with no loss in system performance.

3. Future design - If the bus is greatly overdesigned for the demands of the processors, savings could be made on a more limited bus or more processors could be added for better performance.
- If the bus traffic is too close to its practical limit, an improved bus should be considered for future systems.
- If there is an arbitration algorithm implemented in hardware that is blocking some of the processors from reasonable bus access, an improved algorithm should be implemented.

11.2 Bus Saturation vs. Time

Measurement technique

The method of determination of the point of saturation is similar to that for switching networks; introduce artificial traffic requests at an increasing rate until the actual traffic on the bus shows little or no increase. Several different types and patterns of traffic should be used to reduce biases associated with particular types. The level of traffic during actual system operation can then be observed to determine whether it is near saturation.

Possible uses

1. Evaluation - As for switching networks, a general bus design has a practical traffic limit that is below the roughly calculated theoretical maximum. This is a function of number of bit paths for instructions and data, transmission length, multiplexing (if any), bit rate, arbitration, handshaking, propagation delays, synchronization of the processors, error detection, collision avoidance, etc. Saturation is the point at which an increasing rate of bus requests leads to only a slight increase or even a decrease in the useful traffic on the bus. It is useful to know this point because it is the traffic limitation first encountered and is therefore the deciding one.
- Once the saturation point is known, one can observe a running system and determine how close it is to the saturation point. The exact level of saturation is seldom precisely defined, and may vary with changes in the above factors plus patterns of access requests. In most bus systems, however, many of the parameters are fixed, and the saturation point can be expected to remain within a fairly narrow range. Therefore a determination of the degree of saturation will generally be accurate enough to be useful.

2. Tuning - If the bus is carrying near-saturation traffic loads, there is generally a degradation or risk of degradation in performance. It could therefore be useful in such cases to reduce the traffic by changing the timing of requests, etc.

3. Future design - If the saturation point of the bus is too low to provide adequate service to the processors, the bus should be redesigned (or possibly replaced with some other form of interconnection such as a network). Possible ways to raise the saturation point of the bus include increasing the width of the data path, avoiding multiplexing of instructions and data, increasing the bit rate, reducing propagation delays by shortening the bus, making handshaking and arbitration more straightforward, tuning hardware

access opportunities to propagation delays, designing multiple parallel buses, etc.

11.3 Bus Access Delays

Measurement technique

To measure an overall access delay from the processor bus interface may be rather difficult in non-queueing systems. There are usually signals from the processor indicating waiting intervals, and these can be used when correlated with bus access requests. If queueing of requests is used, it may be possible to look at the queues in order to determine how long requests remain there.

Possible uses

1. Evaluation - Multiprocessor systems with bus interconnection generally use some sort of arbitration scheme to determine in a fair manner who is eligible to use the system bus next. One popular approach is a central arbiter that accepts requests from all sending devices on the bus and determines the winner. This works well because the bus (and therefore the maximum propagation time) is usually very short. Distributed arbitration is also used, in which, for instance, each processor keeps track of what has been accessed and by whom, and knows who will have priority next. Time-slot allocation can be used, but with a large number of processors will waste an unacceptable amount of bus bandwidth. It is theoretically possible to use the "carrier sense multiple access with collision detection" algorithm found in many serial distributed networks, but it is considered to perform poorly under heavy traffic loads, as are often found on a system bus, if there are noticeable propagation delays and long headers.

At any point where several devices contend for one resource, there are likely to be access delays, in which one device has use of the resouce while the others wait. There are two major contention-handling mechanisms: the request signals are queued, or discarded. If the requests are discarded, the requesting device must make note of the fact, and reissue requests as needed. If they are queued, the requesting device normally need only wait until there is a response (though some provision must be made for overflow of the queues).

Access delays are largely a function of system loading and are therefore highly variable. Since they strongly influence system performance and since they are dependent upon the architecture and the set of jobs being run, it is important to be able to observe the frequency of and distribution of time penalties for various types of access delays in order to determine how well the hardware and software work together.

2. Tuning - If the request rejection method of arbitration is used, it may be practical to modify the timing of the retry algorithm so there will be less contention on successive attempts.
- If certain devices are delayed more than others, one may wish to change the arbitration algorithm, the job involved with the device, or the processor running the job.

3. Future design - If a number of delays are observed as a result of the presence or absence of queues, there can be an incentive to switch from one request rejection technique to another.
- Certain hardware changes could make the distribution of delays among processors more fair.

12. QUEUES

Many of the functions that have to be performed in a multiprocessor system make use of queues. Queues can replace 'reject and retry' techniques, which is useful in many cases and vital in some. They can be used in system communications, either in system interfaces of each processor, or in nodes of the communication system. Shared memory resources may have access queues, as may I/O interfaces. Arbitration for access to the system bus or interconnection network may also be handled in this manner. There are a great number of software queues used in most systems, including lists of tasks waiting to be executed.

12.1 Queue Lengths vs. Time

Measurement technique

Although there may be pointers that can be scanned by software, it is necessary to use hardware methods to avoid perturbing the system being measured, unless some way can be found to make use of otherwise unused processor cycles with "acceptably low perturbation" of the system. This can often be accomplished by monitoring reads from and writes to memory areas used for queues. A similar technique can be used with hardware first-in, first-out (FIFO) integrated circuits used for queues. The length profile for each queue must be maintained separately.

Possible uses

1. Evaluation - Tracking the lengths of queues as a function of time can be useful both for observing the general performance of the system and for looking for particular types of problems. When a given queue is empty, the associated device is generally idle (or working on the element most recently in the queue). When the queue is fairly short, the device is working steadily without much delay for service of the elements in the queue. When the queue is longer, there will be significant service waiting times. If the queue is filled to overflowing, there will be errors unless there is provision to halt processes attempting to add items to the full queue.

Queue length is a function of the state of the system at any given time and its history, and therefore varies as a function of these parameters. If there is a time interval over which one or more queues have problems that affect performance, it should be possible to trace their causes by looking at the preceeding events.

2. Tuning - Measurements may identify certain features of the software which result in excessive queue lengths. If so, one can often change the software to reduce these occurrences.

3. Future design - To reduce overflow of queues, it may be necessary to make the queues longer. Multiple servers for single queues can frequently help.

12.2 Queue Lengths During Benchmark Execution

Measurement technique

Observation of queue lengths during the execution of benchmark programs must be carried out. It is important to be able to relate the queue lengths to progress through execution of the program. Thus queue length should be related to instruction addresses. This may eliminate the need to record time stamps for this measurement.

Possible uses

1. Evaluation - This measurement is very similar to the one for queue lengths as a function of time, but gives information pertaining to specific activities associated with specific points in the benchmark. This may make it easier to identify actions such as bus uses, memory accesses, and communications attempts which cause queueing problems.

2. Tuning - Observing the effects on queues of various combinations of activities, one may be able to develop rules of thumb on which techniques to use and which to avoid in designing software for a particular architecture.

12.3 Distribution of Queue Lengths

Measurement technique

The above two measurements may be displayed as *statistical distributions* as well. Actually, if only the statistical distribution is required, recording the statistical information in real time would be easier, requiring only a set of registers for the data on each queue, rather than a large block of memory with timestamps or instruction addresses.

Possible uses

1. Evaluation - This measurement is a subset of the other two queue length measurements, and may be easier to obtain and record than the others. It should be useful in looking at the general performance of the system and for spotting potential trouble spots such as bottlenecks in the architecture. One of the specific uses could be inspecting the software-related bottlenecks in parallel programs such as critical nonparallel sections. The detailed queue length measurements can then be made of only the limited number of queues known to be trouble spots.

2. Tuning - It may be possible to "evolve" parallel programs from normal sequential programs (already existing or written for the purpose), by gradually "parallelizing" various parts of the program, and looking at the distributions of queue lengths in the task synchronization mechanisms to determine which areas to concentrate on next.

3. Future design - Same as above.

13. PROCESSES

13.1 Time Spent Waiting in a Job Queue

Measurement technique

If the job queue is handled in software, one should be able to keep track of the time-in-queue by tracing addresses, memory pointers, etc. These can be observed by monitoring the local bus lines. If the queue is handled in hardware, similar techniques will work.

Possible uses

1. Evaluation - The tasks representing waiting processes generally organize them as a hardware- or software-based job queue. They therefore are treated like other queued elements, and the implications of their queueing behavior are similar. The length of the job queue and the average waiting time are critical parameters because they are closely tied in with system performance. A long waiting time implies that the processing capabilities of the system may not be sufficient for the demands of the processes, either in general or for this particular set of jobs.
Another possibility is that the job has insufficient parallelism to utilize the resources at hand.

2. Tuning - If the processing load of a given set of tasks is such that there are long delays in the job queue, leading to sluggish performance for time-critical jobs, a new set of priorities can be established, or the load can be decreased until there is adequate performance in these areas.

3. Future design - If the job queue is generally too long, more processing power in the system or enhanced hardware-based job queueing should alleviate the problem in most cases.

13.2 Time Spent Waiting for Shared Resources and Sync Signals

Measurement technique

This is a measure of the time spent waiting in queues for shared resources and synchronization signals. These queues differ from others only in their assignment, not in the techniques used. Thus the measurement techniques discussed above are applicable.

Possible uses

1. Evaluation - Shared resources can include common memory, disks, I/O devices, etc. Synchronization signals are physical signals, memory locations, etc., often dealing with "tokens", which allow the various parts of a parallel program to keep in step. Once activated, processes contend with one another to get access to shared resources, and periodically stop at various points in the execution, waiting until they can successfully send or receive synchronization signals. The waiting time for these two types of access is added directly to the execution times of the processes, so minimization of these delays is essential for good system performance. If waiting times are too long in a variety of situations, then either improvement in the basic design of the mechanisms and their support structures is needed, or the demands exceed the basic capacity of the system.

2. Tuning - A wide variety of things can be done to help reduce contention for shared resources and smooth the operation of synchronization signals, many of which have already been discussed. The exact methods used should depend on the situation.

3. Future design - Approaches such as improved caches can help to minimize contention for resources such as shared memory. Inceased connectivity can improve other architectures. Improved hardware support for synchronization mechanisms can make them work more efficiently.

14. VARIABLES

14.1 Relative Number of Local, Global, Variables

Measurement technique

The number, or relative number (mix), of local and global variables can usually be determined by the compiler or loader. If some of the variables are in complex data structures, it may be necessary to simulate execution of the programs to observe the nature of the variables, though conservatively classifying all questionable variables as global could get around this problem. A good general rule of thumb is that unrelated programs on a multiprocessor system usually do not share any non-system variables.

Possible uses

1. Evaluation - The number, or relative number, of local and global variables is of interest in any multiprocessor system. Many systems have some memory local to each processor, in which local variables are placed for quick access. Global variables are usually placed in a shared memory, and contention among processors for access can create significant delays. Global variables are often used for synchronization of multiple processes, in which case some contention and delay due to waiting are unavoidable.

2. Tuning - If there are provisions in a system for efficient local variables, and if some existing variables are unnecessarily global, conversion to local can help to reduce system resource contention and improve performance.

3. Future design - The number, or relative number, of local and global variables can be used to decide whether special treatment of certain types of variables is warranted for future designs. If it turns out that many global variables could be local and thereby improve efficiency, except that there are no hardware facilities provided for variables local to each processor, there would be an incentive to design such facilities. One objection to such an arrangement has been that is not known in advance what size to make local memory. With this type of analysis of many programs, one can obtain a general idea of what size is needed, which in turn could guide a conservative design to meet the expected needs.

Another approach is to make the mix between local and global memory dynamically variable.

14.2 Variables Transferred Between Processes

Measurement technique

Since the number of variables transferred between processes relies partly on the real-time interactions of the processes, the number of interchanges is usually not readily predictable by analysis of the program. Fortunately, it should generally be relatively straightforward to monitor such transactions on the bus/switching network. They can be identified as writes and reads in shared memory, or transfers over a switching or intercommunications network.

Possible uses

1. Evaluation - The variables that may be transmitted between processes include data and synchronization signals. Such transmissions, possibly along with I/O and shared memory transfers, make up the traffic that fills the system bus/switching network. This is commonly the bottleneck in multiprocessor systems. The transfers may be direct, or through main memory (the normal method when the two processes run sequentially on the same processor), which would require a double transfer. In computing the effect on traffic, any handshaking must also be taken into account.

2. Tuning - If the programmer has a choice among synchronization mechanisms (as is often the case), and one method is observed to generate less traffic than the others, that method should be used wherever other factors do not outweigh minimization of traffic.
- The pattern by which programs are broken down into parallel tasks can be chosen in a manner that minimizes the number of necessary interactions among the processes.

3. Future design - Hardware support for more efficient (or more parallel) interchange of synchronization signals and other variables should be developed as needed.

14.3 Parallel Processes Sharing Each Variable

Measurement technique

The number of parallel processes sharing each variable can usually be derived by analysis of the program, without any actual execution being required.

Possible uses

1. Evaluation - A variable shared among several processes can be a data element, or it can be in use for synchronization of the processes. If it is a data element, it serves as a means of communication for sequential processes, and it may very well be a source of contention among parallel processes. If it is used for synchronization, it shows how the various processes are interrelated.

2. Tuning - To allow for the greatest possible parallelism, the number of shared variables linking together a group of processes should generally be minimized. An abundance of shared variables usually means that sychronization delays will occur.

3. Future design - There may be some incentive to develop a specialized memory service (i.e.

multiport memory, etc.) dedicated to shared variables or a subset of shared variables. If this were done, one would need to know the magnitude of the service required.

14.4 Placement for Efficient Caching

Measurement technique

In systems employing cache memories, contiguous placement of variables usually results in more efficient cache operation. The compiler and linker should produce an output by which one can determine the addresses and thus the groupings of the variables. The sequence of their use may be obvious from the program, or execution-time measurement may be necessary. The cache hit ratio, described above, is another measure of the success of variable placement.

Possible uses

1. Evaluation - By the structure inherent in most cache memory designs, memory elements that are close together in the memory address range are handled as a group more quickly and efficiently than those that are scattered through the address space. This is because: 1) cache transfers are in blocks, which may contain more than one variable, 2) a set of needed variables is more likely to fit into the cache if the variables are adjacent in memory rather than spread out, so the hit ratio should be higher, and 3) a moderate number of variables grouped together have less chance of interfering with one another (by existing in blocks with the same higher address bits which would cause unwanted replacements) than if they are spread out.

2. Tuning - It should usually be possible to arrange the addresses of the variables so they can be handled more efficiently by the cache.

3. Future design - Knowledge of the address groupings of variables can aid in the design of caches. Knowledge, by the compiler, of the caching algorithm should improve the execution speed of the generated code.

15. INSTRUCTIONS

15.1 Size of Each Task in Memory

Measurement technique

The size of each task in memory can almost certainly be determined at compile time by analysis of the compiler output.

Possible uses

1. Evaluation - To promote efficiency with the way in which contemporary systems handle memory structures, tasks are usually set up in memory as single blocks or small numbers of blocks. It therefore makes sense to think of a task as having a certain size and taking up a certain amount of memory space. This is a useful concept when dealing with multiprocessor systems, which can have local memory or cache memory in which to store local data and instructions. The size of the tasks (among other things) determines whether one, or possibly several, can be kept in local storage to reduce the need for shared memory or interprocessor accesses. Some compilers produce much more compact code than others. This is therefore a useful tool in the evaluation of compilers.

2. Tuning - If the individual tasks take up too much room to fit in local storage and operate with the greatest efficiency, perhaps they can be broken up into smaller tasks that fit better.

3. Future design - Better compilers can be designed, or larger blocks of local storage provided.

15.2 Is Shareable Code Shared?

Measurement technique

The degree of code sharing may be derived from the output of the compiler. (This assumes that the operating system allows code sharing.)

Possible uses

1. Evaluation - In the hierarchy of called and calling processes, some compilers will place independent tree structures under each task. In other words, two tasks of equal status in the hierarchy which both happen to call the same subtask will each be given a separate copy of the subtask. This causes inefficient use of memory, but may reduce the extent to which the tasks interfere with one another. Other compilers will sometimes find subtasks that are used by several tasks and link all the calling tasks to the same subtask. This tends to use memory more efficiently, and the use of caching can limit shared-resource contention. There is probably an optimal tradeoff between the two techniques. The degree of sharing shows what tradeoffs were made in the design of the compiler. This same argument can be applied to smaller pieces of code as well.

2. Tuning - If the amount of code sharing is adjustable by the programmer, it can be advantageous to choose the pattern of usage for the particular program.

3. Future design - It may be a good idea to implement adjustable code sharing for systems that do not already have it. It is not immediately clear whether software support alone is sufficient or whether some sort of hardware support is also needed.

15.3 Execution Path Through Instructions

Measurement technique

With access to the address and data buses, along with control signals identifying instruction fetches, one can determine what is being executed in the program. Logic analyzers are usually used for this purpose. The point at which to start recording is often detected using an address recognizer, though a signal instruction inserted in the code will also work. Since modern processors are very fast, have long addresses, and execute long programs, it is difficult to store complete histories in full detail. Having multiple processors only compounds the problem of too high a rate of data collection. A variety of methods may be used to get around this difficulty, including ignoring some of the addresses (i.e., looking only for jumps), looking only at short intervals, and using very high speed and expensive measuring equipment.

Possible uses

1. Evaluation - It can be very useful to know the path of execution taken through a program for a variety of reasons. First, it can help one to determine whether a program is operating correctly, by locating the point of deviation from the expected path. Second, it can be measured and recorded with timestamps, to observe speed of execution and analyze synchronization mechanisms in action. Third, it allows one to determine what decisions were made at branches, how many times loops were executed, etc.

2. Tuning - Points at which the program does not conform to the desired algorithm can be identified and fixed.

3. Future design - In view of its usefulness, systems should provide signals that simplify the acquisition of records of the path of execution.

15.4 Instruction Placement for Efficient Caching

Measurement technique

In systems with caches, measurements must be made to determine if instructions are placed in memory so as to allow efficient caching. Normally the memory allocation can be determined after a program has been compiled and linked. However, only by knowing the instruction execution sequence (the above measurement), the caching scheme, and the resulting hit ratio can one make conclusions on the success with which instruction placement fits with the caching system.

Possible uses

1. Evaluation - With most modern cache designs, there are two situations which result in inefficient caching. These are unnecessary scattering of the instructions in memory, so that the code will not fit well in cache, and placement of tasks which are used alternately in blocks of memory with the same high-order address bits, which causes them to repeatedly replace one another if there is an insufficient number of sets in cache to contain all the conflicting tasks.

2. Tuning - If the programmer has any capability to override the allocation of program memory, it may be advisable to compensate for poor automatic allocation choices.

3. Future design - Compilers should be developed that take advantage of the capabilities of caches and other nonstandard memory elements. The compiler must be aware of the caching scheme. Caches should be designed able to hold all the material necessary for typical groups of concurrent tasks, where

other factors do not intervene.

15.5 Instruction Execution Times

Measurement technique

Because of pipelining or instruction queueing, observing the time interval between successive instruction fetches may not produce accurate measurements of instruction execution times. Techniques include looking for memory addresses to be accessed following the instructions, putting instructions in loops, or external simulation of the instruction queue.

Possible uses

1. Evaluation - This measurement refers to the average total time taken under a stated set of conditions for a given instruction with a given set of parameters (if various parameter types are permitted) to complete execution, measured from the point at which execution begins. Since different processors execute instructions in different ways, and since many of the instructions require various numbers of memory accesses, a complete evaluation will be expensive and difficult. Nevertheless, an effort should be made since the overall speed of execution of instructions is a basic determinant in the performance of the system.

2. Tuning - In writing a program, there is often a choice among methods which use different instructions. If some instructions have a clear time disadvantage compared to others, procedures which use these instructions should be avoided whenever possible.

3. Future design - If certain needed instructions exhibit poor performance in a multiprocessor environment, the hardware should be changed so performance is improved for these instructions.

16. SHARED RESOURCES (may include memory)

16.1 Fair Sharing of Resources

Measurement technique

Shared resource requests and resource accesses among the processors are measured to determine if all processors get a fair chance for resource access. It will be necessary to observe resource access requests and access grants, and record the number of requests and the delays before grants, with timestamps. Requests and grants are often represented as single-wire signals. Since such requests may be queued or discarded and then resubmitted, the exact interpretation of such measurements depends on the particular system being tested.

Possible uses

1. Evaluation - Some shared resource multiprocessor systems have no explicit form of access control designed to allow fair access to the resources by all the processors, relying on random timing of access requests to accomplish this function. Most modern designs, however, have a built-in mechanism intended to guarantee some degree of fairness. In either case, there must be something to prevent more than the maximum permitted number of processors (usually one) to communicate with any given resource at any given time. This measurement evaluates the success in achieving the desired result.

If there is no built-in fairness, the degree of fairness is completely dependent on the statistics of the timing of access requests, which in turn are partly a function of the characteristics of the processors, the resources, and the current programming load. If enough information on these characteristics is available at the time of the design, the designer may be justified in specifying a system without built-in fairness.

With a fairness mechanism, the observed degree of fairness is also variable, but usually to a lesser degree. There is a range in emphasis on fairness, reflecting a variety of definitions of fairness, from "no processor should be locked out permanently" to "all processors should have approximately equal access opportunity". Different program loads also affect fairness.

Fairness can have varying degrees of importance depending on the work being done. If there are certain processors which get little access to needed shared resources, and if these procssors are running only low-priority background tasks, then there is little harm as long as their associated processes are not stopped completely. If some of the processors working on a highly parallel problem do not get fair access, then the entire job may be slowed significantly. If, through some fault in the design, some of the processors can not get access to a synchronization mechanism, then a deadlock may result.

2. Tuning - If there are problems arising from lack of fairness in a system, this measurement may identify software approaches by which the problems can be solved or circumvented.

3. Future design - The degree of fairness is strongly dependent on the details of the arbitration method used. Priority for a system bus, for example, may be fixed for each processor, which does not guarantee fairness, or may use round-robin arbitration, which virtually assures some degree of fairness. Such considerations should be taken into account when designing the hardware support for arbitration.

16.2 Distribution of Processors Waiting

Measurement technique

The distribution over time of the number of processors waiting for a resource should be measured. This must be possible for any particular resource and for all resources. Access requests can be detected on the system bus/network or at the location of each requested resource. This information can be recorded with timestamps and compared with a similar record of access grants. The exact method of comparison depends on the contention-resolving mechanisms of each system.

Possible uses

1. Evaluation - "Waiting for a resource" could be interpreted to mean a variety of different things, from "inactive until resource granted or timeout" to "working on an alternative process while the primary process is blocked, due to unfulfilled access request". In any event, a processor that is waiting for a resource suffers some loss of useful execution time, ranging from a few clock cycles to the entire duration of the wait. It is therefore prudent to make some effort (the level dependent on the penalty of waiting) to minimize the amount of unintentional waiting.

Some resources may be more heavily used than others, and are therefore the cause of more processor waiting time than others. It would be useful to identify these resources and determine why they are thus distinguished.

2. Tuning - It may be possible to redistribute the resource demand of concurrent processes so they do not have to wait so long for resources.

3. Future design - If some of the resources that are most heavily in demand can be "expanded" into multiple resources, there may be a reduced amount of waiting.

16.3 Complete Execution Statistics

Measurement technique

Statistics on addresses, ranges of addresses, reads/writes, instructions/data, etc., must be recorded. Observation of address accesses can be made by monitoring the address buses of the processors, and the system address bus or packet addresses. Virtual addresses would generally be used for evaluation of the software; physical addresses would be used in anaylsis of the system hardware. The method of recording depends on the system and the specific type of measurement desired.

This measurement will generate a vast volume of data, a sample point for each instruction executed by each processor. For this reason it must be used sparingly, or for only short code segments or the recording system will be overloaded. This is a particularly useful place to apply the Transaction Analyzer [GAUD] approach to allow "front end filtering" to reduce the data recording requirements. If predictable instruction sequences can be recorded in a "shorthand" form, the limited recording bandwidth can be largely reserved for the unpredicted actions.

Possible uses

1. Evaluation - This information refers, by memory addresses, to access for the fetching of instructions. Similar data can be kept on data accesses. Knowledge of which addresses are accessed and with what frequency can allow one to debug programs and locate hardware and software bottlenecks.

2. Tuning - The software-related bottlenecks associated with a given program can often be eliminated by rewriting the sections of the code that cause them.
- The efficiency and the nature of memory utilization can be determined and adjusted.

3. Future design - Many features of the hardware design that produce or contribute to the formation

of bottlenecks can be detected and eliminated.

16.4 Bandwidth Used to Wait for Resource

Measurement technique

In order to determine how much interprocessor communication bandwidth the "waiting for shared resource" (say a spinlock) function uses, the addresses associated with such functions can be found at compile time. Access requests to these addresses can be recorded, and, with the overhead of each taken into account, compared to the calculated maximum bandwidth. Analagous mechanisms exist in other architectures, such as the bandwidth taken by synchronization packets.

Possible uses

1. Evaluation - In many systems the method of waiting for a shared resource involves repeatedly polling a given memory location until it contains a desired value. This can have the side effect of using up a large part of the access bandwidth to the shared resource, making it unavailable to other processors and possibly slowing down all communications. Fortunately, one can often use techniques to minimize this effect. The degree to which such traffic can be reduced determines the number of processors that can be added to the system with close to linear speedup. The degree to which this is a problem should therefore be observed.

2. Tuning - Some functions of this type are software controlled, and as such should be adjusted for minimum bandwidth utilization with minimum delay in detection of resource availability.
- The need for accesses that involve waiting should be minimized in programs.

3. Future design - Devices such as local caches with controllers that monitor the bus can reduce the bus bandwidth utilization of this type of function dramatically while maintaining good response time. These and other devices, properly tuned, can greatly improve the performance of multiprocessor systems.

17. OWNERSHIP TOKENS

17.1 Time Spent Processing Tokens

Measurement technique

This is a measurement of the amount of time spent processing and transferring ownership tokens. References to tokens can often be detected by observation of addresses or sequences of addresses, or the data written into these addresses. The elapsed time to check/write these tokens is accumulated.

Possible uses

1. Evaluation - There are many shared resources which are best adapted to the concept of ownership. For instance, a disk drive set up to transfer a particular file can not quickly be changed to do something different until it has finished its present task. Therefore, to assign ownership to the processor with which it is working would be reasonable. The same logic could be applied to a user interface. Common memory is sometimes divided into partitions, with particular processors as owners of the various partitions at any given time. This is usually acieved with hardware arbitration schemes, or memory management units, rather than tokens.

For all of these cases, ownership is often signified by possession (local existence) of a value called a token. Transfer of ownership is accomplished by moving the token from one location to another, hopefully with handshaking to assure that there is agreement on the location of the token. Since this involves work on the part of the processor-system interfaces and presumably the processors themselves, there is some performance penalty for the time taken up working with tokens. This is a function of average transfer rate, frequency of transfer, and the necessity of looking for the presence of a token before performing certain functions.

2. Tuning - Ownership should not be transferred more often than necessary, in order to minimize this overhead.

3. Future design - The delays from the necessary checking for tokens should be made as small as possible. The time to transfer ownership should be kept small, with the tradeoff of keeping it reliable.

17.2 Effect of Token Loss

Measurement technique

Token loss is usually so serious that great design care has been taken to avoid its occurrance. If possible, a characterization should be formed of the nature and pattern of token losses. The effects of token loss are best observed by artificially inducing destruction of tokens.

Possible uses

1. Evaluation - If a token is lost during transfer, it may appear to the processors that nobody owns the associated resource, possibly resulting in permanent deadlock. A token might also be deliberately destroyed as part of a test, or with the intent of making the system inoperative. Handshaking and other approaches are used to make the loss of a token as unlikely as possible, but such losses can not be prevented entirely. There should therefore be some mechanism to handle the loss of tokens.

Simple recovery mechanisms might restore operability but result in the loss of some or all of the currently executing processes. A system reset, for example, would restore the tokens but would also destroy the current programs. More sophisticated designs could in many cases restore the tokens without disturbing the programs.

Because of the importance of tokens and the variety of ways in which they are treated, it is important to observe the token recovery capabilities of any machine to be evaluated.

3. Future design - It should be difficult or impossible for normal users to deliberately alter tokens, in any system that is intended to be reliable. Observation of current token handlers can help in the design

of new ones.
-The token recovery mechanism should work with minimum impact to the normal operation of the system.

18. SYNCHRONIZATION

18.1 Interprocess/Intertask Synchronization Cost

Measurement technique

This is a measure of the amount of time taken for an interprocess or intertask synchronization. Examination of the compiler or loader output can identify instructions accessed immediately before synchronization attempts, and the ones immediately after success. The interval between addressing these locations must be recorded. An alternative measurement technique could use the flow of tokens in specific memory locations or interprocessor paths.

Possible uses

1. Evaluation - The frequency of attempted synchronizations and the average response delay show how well the mechanism is fitted to the needs of the system.

2. Tuning - In dividing the program up into tasks, the costs of interaction among the processes must be taken into account.

3. Future design - Since some sort of synchronization mechanism is necessary in multiprocessing applications, maximizing the speed of the actual synchronization (not counting time spent waiting for it to begin) can be very important to overall system performance. The time is highly dependent on the mechanism used. In any case it should be small fraction of total elapsed time.

18.2 Is Static Task Allocation Correct?

Measurement technique

In systems using static allocation of resources, one must determine how efficiently the timing of the tasks is tuned, to allow efficient use of the processors and quick execution of the programs. The information needed for analysis of performance can be derived by measuring the summed processor active time and the summed processor inactive time, possibly making allowances for synchronization and communication time. Performance with a given allocation, as compared to performance of the same job with a different allocation, gives an idea of the appropriateness of the allocation.

Possible uses

1. Evaluation - Many multiprocessor systems use dynamic allocation of tasks, in which any processor completing its current task automatically switches execution to the next task waiting in the job queue. Aside from the possible problem of different communications latencies among different sets of processors, this approach usually does a good job of making efficient use of the capacity of the processors. In some systems, however, the entire program will run sequentially on one processor unless the programmer deliberately directs various tasks to different processors. To do this well in a situation where there may be a good deal of communication among the processes, the programmer must have a fairly good idea of how long it will take each process to run, and allocate the tasks to the various processors so that most of the processors will be doing useful work most of the time. This is obviously a very complex and difficult procedure.

In other systems, the compiler may decide on allocation of the tasks ahead of time. Obviously, the compiler must have good estimates of the execution times in order for this to work well. In either case, records of processor usage with several different sets of programs can show how well the allocation scheme works.

2. Tuning - Several iterations of allocation of the tasks, with knowledge of the efficiencies of previous versions, are essential to optimized programming on this type of system.

3. Future design - One should at least consider implementation of dynamic allocation of tasks wherever possible, both for more efficient execution of programs and for simpler applications programming. Some machines currently in existence combine the properties and the advantages of static and dynamic allocation. Normally dynamic allocation is used, but the programmer can specify that certain tasks are to be run on certain machine resources, in order to minimize communications paths or make use of processor-dependent I/O devices.

19. PRIORITY

19.1 Effects of Priority Scheme

Measurement technique

This parameter concerns the effects of various task execution and resource access priority (or other scheduling) schemes for general resolution of resource contention, on overall system performance and on individual performances of high priority and low priority tasks. If the priority techniques are software selectable, measurement and comparison should be fairly straightforword. If they are not software selectable in the machine of interest, simulations or hardware modifications may be in order.

Possible uses

1. Evaluation - In a system with multiple processors and multiple tasks running simultaneously, there is often the question of what should run first or which processor should have first access to resources. This problem is resolved according to the type of priority scheme used. Fair access does not always mean equal access. Sometimes all tasks are given the same priority, and resolution left to chance timing. In other situations, a strict hierarchy is set up, with high priority tasks crowding out others. Since the results of these choices can have significant effects on observed system performance, their influence should be closely observed.

2. Tuning - If the priority scheme can be chosen in software, the technique that appears to give the best performance should be selected.

Many systems have some sort of software control over priority. The effects of priority on execution of a given set of tasks can then be observed.

3. Future design - The priority techniques that work best for the application at hand can be designed into systems, with hardware support for fastest operation. It may be a good idea to allow software selection of priority for systems that do not currently have it.

20. FAULTS

20.1 Detection and Logging of Faults

Measurement technique

Erroneous operation of the system must be detected and logged. If the desired patterns of execution are predicatable, instruction address traces should provide clues on points of departure from expected execution. The Transaction Analyzer [GAUD] is a preferred instrument for this application.

Possible uses

1. Evaluation - The analytical equipment should be able to detect and record a wide variety of faults. The existence of faults should be seriously considered in system evaluation. Some systems may be able to recover from faults without erroneous results.

2. Tuning - If there are a large number of faults, or if they are not being effectively detected and overcome by the system, improvements to the reliability mechanisms should be incorporated in the software (and the hardware) of the system.

3. Future design - Systems should be designed so that signals indicating faults can be accessed by exter-

nal equipment, particularly those fults that prove to be of the most interest.

20.2 Response to Faults

Measurement technique

This involves observation of responses of the system to naturally and artificially induced faults (fault tolerance). To test fault handling of the system, known faults can be introduced and the system response recorded.

Possible uses

1. Evaluation - The system should be able to detect faults, correct them, and make records of them. If this is not done, the system is not reliable.

2. Tuning - If there are numerous faults, or if they are not being effectively detected and handled by the system, additional reliability mechanisms must be incorporated in the software.

3. Future design - If current designs do not provide adequate fault protection, hardware support for fault detection and correction should be introduced.

21. SIMULTANEITY

Measurement technique

This parameter is a consideration in the development of measurment techniques, not in the operation of the machine under test. It concerns whether measurements that are intended to be taken in parallel are simultaneous within sufficiently close tolerances so as to insure correct interpretation of the results. The measurement may best be done by the use of a small number of fast, expensive devices such as high-speed oscilloscopes, to calibrate a multitude of somewhat slower, less expensive probes.

Possible uses

1. Evaluation - One of the most fundamental measurements made when analyzing a system is to observe some event taking place and to determine what other events are taking place at the same time. Since many events within a system are very fleeting, the question of simultaneity of the various measurements becomes extremely important. The precision of the measurement can not be better than the possible skew in the times of measurement. This problem is highlighted in the recording of sequences of events. For a given possible skew, the order of events can not be reliably determined if the two events are separated by less than a certain interval.

2. Tuning - With a given amount of possible skew, it is important that no weight be placed on the validity of measurements requiring a smaller skew.
- If there is a known maximum possible skew with a large fixed component, such as propagation delay, one can often factor out this component and deal only with the variable component, thus allowing some types of measurement to be more precise (and hopefully more accurate) than the overall skew would

suggest.

3. Future design - New designs may have measuring equipment imbedded in the computer system as delivered. Thus the system and external measuring equipment will work together, with specialized hardware in each, so that there will be fewer inaccuracies in the timing of measurements. An example of such a system modification would be to have certain useful timing signals made available to probes.

22. SUMMARY

Measures involving quantities may be taken and evaluated as specific values during particular samples, continuous samples over time, intermittent samples over time, or statistical samples such as averages and distributions. Values may be exact or classified in ranges. More than one of these forms of measurement may be used for the same parameter, when appropriate for a particular application.

The above comments are also mostly applicable to time measurements themselves. It is important that the time delays involved in the measurement process not lead to inaccuracies in the measurements.

Some of the "measurements" described herein may be made by analysis of the object code or compiler output. Most must be made on actual operating hardware. Some of the measurements may require hardware or software modifications. Some may not be practical on any currently existing machine.

This list attempts to describe measurements that may be useful for evaluation, tuning or future design efforts in the major areas of knowledge of multiprocessors. Some of the measurements may be trivially easy, while others may turn out to be impossible.

23. ACKNOWLEDGEMENT

Thanks are due Robert J. Carpenter for valuable discussions and editorial suggestions, and to Bill Burr, Philip Gaudette, and Alan Mink, for their help in reviewing the draft version.

24. REFERENCES

[GAUD] Gaudette, Philip and Robert B. J. Warnar, "A Fast Loadable Hardware Interpreter for Finite State Automata" NBS Internal Communication, 1984.

25. BIBLIOGRAPHY

These make up a small subset of all the papers that have been written on multiprocessors and multiprocessor measurement. Because of the rapid rate of advancement in this area of study, much of the most recent research information exists only in the form of unpublished papers and sets of transparencies. For updates on the work currently being done, one should consult the researchers at universities and at private and government research centers.

Agrawala, Ashok K.; Edward G. Coffman, Jr.; Michael R. Garey; Satish K. Tripathi, "A Stochastic Optimization Algorithm Minimizing Expected Flow Times on Uniform Processors", *IEEE Transactions on Computers*, Vol. C-33, No. 4, April, 1984, pp. 351-356.

Barak, Amnon and Zvi Drezner, "Distributed Algorithm for the Average Load of a Multicomputer," Computing Research Laboratory TR-17-84, The University of Michigan, Ann Arbor, Michigan, March, 1984, 24 pages.

Briggs, Faye A. and Michel Dubois, "Effectiveness of Private Caches in Multiprocessor Systems with Parallel-Pipelined Memories", *IEEE Transactions on Computers*, Vol. C-32, No. 1, January, 1983, pp. 49-59.

Browne, J.C., "Issue Summary from the NYU Workshop on Parallel Computing," University of Texas at Austin, April 24, 1982.

Buehrer, Richard E.; Hans-Joerg Brundiers; Hans Benz; Bernard Bron; Hansmartin Friess; Walter Haelg; Hans Juergen Halin; Anders Isacson; Milan Tadian, "The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System", *IEEE Transactions on Computers*, Vol. C-31, No. 11, November, 1982, pp. 1035-1044.

"Building a Parallel Computer Means Starting All Over Again", *The Economist*, March 2, 1985, pp. 83-84.

Dietz, William B. and Leland Szwerenko, "Architectural Efficiency Measures: An Overview of Three Studies", *Computer*, Vol. 12, No. 4, April, 1979, pp. 26-33.

Flynn, Michael J. and Lee W. Hoevel, "Measures of Ideal Execution Architectures", *IBM Journal of Research and Development*, Vol. 28, No. 4, July, 1984, pp. 356-369.

Folsom, Virginia L. and Elizabeth I. Olsen, editors, "Bibliography of Computing Research Laboratory Reports, 1982-1983," Computing Research Laboratory TR-1-84, The University of Michigan, Ann Arbor, Michigan, January, 1984, 33 pages. (For further reading.)

Fromm, Hansjorg; Uwe Hercksen; Ulrich Herzog; Karl-Heinz John; Rainer Klar; Wolfgang Kleinoder, "Experiences with Performance Measurement and Modeling of a Processor Array", *IEEE Transactions on Computers*, Vol. C-32, No. 1, January, 1983, pp. 15-31.

Kuehn, James T. and Howard J. Siegel, "Simulation Based Performance Measures for SIMD/MIMD Processing", *Computing Structures and Image Processing*, Academic Press, 1985.

Lakshmivarahan, S., "Development of Non-numeric Benchmarks for Parallel Architectures Based On Parallel Sorting Algorithms," private communication, Institute for Computer Sciences and Technology, National Bureau of Standards, June, 1985.

Lubeck, Olaf; James Moore; Raul Mendez, "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810 and Cray X-MP/2," Report No. LA-UR-84-3584, Los Alamos National Laboratory, Los Alamos, New Mexico. (Submitted to *IEEE Computer*).

Marsan, Marco Ajmone; Gianfranco Balbo; Gianni Conti; Francesco Gregoretti, "Modeling Bus Contention and Memory Interference in a Multiprocessor System", *IEEE Transactions on Computers*, Vol. C-32, No. 1, January, 1983, pp. 60-72.

Mink, Alan, "A survey of Multiprocessor Architectures," private communication, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C.

Mudge, T.N.; J.P. Hayes; G.D. Buzzard; D.C. Winsor, "Analysis of Multiple-Bus Interconnection Networks," Computing Research Laboratory TR-12-84, The University of Michigan, Ann Arbor, Michigan, 1984, 17 pages.

Mudge, T.N. and Homoud B. Al-Sadoun, "Memory Interference Models with Variable Connection Time," Computing Research Laboratory TR-16-84, The University of Michigan, Ann Arbor, Michigan, July, 1984, 28 pages.

Reed, Daniel A. and Herbert D. Schwetman, "Cost-Performance Bounds for Multimicrocomputer Networks", *IEEE Transactions on Computers*, Vol. C-32, No. 1, January, 1983, pp. 83-95.

Segall, Zary; Ajay Singh; Richard T. Snodgrass; Anita K. Jones; Daniel P. Siewiorek, "An Integrated Instrumentation Environment for Multiprocessors", *IEEE Transactions on Computers*, Vol. C-32, No. 1, January, 1983, pp 4-14.

Segall, Zary and Larry Rudolph, "PIE - A Programming and Instrumentation Environment for Parallel Processing," Carnegie-Mellon University, April, 1985.

Strecker, William D., "Cache Memories for PDP-11 Family Computers", *Computer Engineering - A DEC View of Hardware Systems Design*, Digital Press, Bedford, Massachusetts, September, 1978, pp. 263-267.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR-85/3296	2. Performing Organ. Report No.	3. Publication Date FEBRUARY 1986

4. TITLE AND SUBTITLE

Performance Measurement Techniques for Multiprocessor Computers

5. AUTHOR(S)

John W. Roberts

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*	7. Contract/Grant No.
NATIONAL BUREAU OF STANDARDS DEPARTMENT OF COMMERCE WASHINGTON, D.C. 20234	8. Type of Report & Period Covered Interim

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

A wide range of possible measures for multiprocessor computers is discussed, along with the realizability of each class of measurement technique and the applicability of the results.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

Computers; measurement; multiprocessor; performance.

13. AVAILABILITY

[✓ xx] Unlimited
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[✓ xx] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

58

15. Price

$9.95