Performance Measurement of a Shared-Memory Multiprocessor Using Hardware Instrumentation

Alan Mink and George Nacht

National Computer and Telecommunications Laboratory
National Institute of Standards and Technology
(formerly National Bureau of Standards)
Gaithersburg, MD 20899

ABSTRACT

A hardware approach is presented for the design of performance measurement instrumentation for a shared-memory, tightly coupled MIMD multiprocessor. The Resource Measurement System (RMS) is a non-intrusive hardware measurement tool used to obtain both trace measurement and resource utilization information. This approach provides more detailed and extensive measurement information than alternative software or hybrid approaches without introducing artifacts into the test results. This is accomplished at a significantly higher tool cost than the alternative software or hybrid approaches. Certain features of todays microprocessors limit the applicability of such a hardware tool. Measurements obtained using this hardware tool on two kernel (small benchmark) routines are presented.

Introduction

In order to correctly conduct performance measurement both data acquisition and stimulus characterization are necessary. Data acquisition is usually the focus of any performance measurement effort, since it involves the actual capture of performance data from the system under test. Stimulus characterization involves understanding the software that is executing on the system under test while the measurement data is being acquired. Stimulus characterization is overlooked in many performance measurement experiments, or is at best relegated to an overall application description such as an editor or compiler. This is insufficient since the measurement data, when analyzed, is used to infer the performance of the system under test without knowing what architectural features the software was exercising.

The acquired performance information can be placed into two orthogonal categories: (1) trace measurement, and (2) resource utilization. Trace measurement is concerned with the activities of the application or system processes, and provides information such as program execution time, execution path, response time, etc. Resource utilization is concerned with the detailed operation of the hardware, and provides information such as cache hit ratios, access delays, duty cycles, etc. Roberts [21] provides a more complete discussion on performance information to be measured.

This National Institute of Standards and Technology contribution is not subject to copyright in the United States. Certain commercial equipment, instruments, or materials may be identified in this paper in order to adequately specify the experimental procedure. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

This work was partially sponsored by the Defense Advanced Research Projects Agency.

Ferrari (8) has classified measurement tools as either hardware, software/firmware, or hybrid. Measurement tools in any of these categories can be used to obtain trace measurement information but hardware tools are required to obtain resource utilization information because the required signals are not visible to software tools. Software and hybrid tools are intrusive in that some additional code is executed either in the measured program or the operating system. This affects the timing characteristics of the measured program. Hardware tools are non-intrusive, as they do not change the operating characteristics of the program.

A common feature of these measurement tools is the triggering facility. A trigger is the mechanism which causes a measurement sample to be taken. Software and hybrid tools use either interrupts or embedded code to cause a trigger, while hardware tools passively monitor a set of signals and cause a trigger when a specific combination of signals occur. Carpenter (5) provides a more complete discussion on the entire range of multiprocessor measurement techniques and their associated trigger mechanisms.

Many measurement techniques are applicable to both uniprocessor and multiprocessor systems. For multiprocessor systems, the process of measurement is more complex due to the additional processors and the cooperating processes they are executing. Measurement tools which cause perturbations in the timing characteristics of these cooperating processes can alter their execution and thus their performance. For this reason, hardware tools which cause no perturbation or hybrid tools which cause minimal perturbation are deemed the most desirable tools for multiprocessor systems. The threshold of perturbation that a given program can tolerate without significantly altering its execution is program dependent. In general, the larger the granularity [14] of each of the cooperating processes and the less frequently communication/synchronization occurs between them, the higher the perturbation tolerance.

Our multiprocessors computational model is one in which a single application program is broken into a number of cooperating processes. These processes are then distributed among the available processors to reduce the execution time. The performance concerns are focused not only on the application program, but also on the multiprocessor architecture and the implementation of that architecture. There is generally a performance hierarchy that starts at the application level. As performance is characterized at that level, there is a desire to delve into the underlying causes of the measured performance. Thus performance is measured from the application, through the operating system, down to the hardware operation. Because of this performance hierarchy, a measurement tool that is capable of acquiring performance information on both trace measurement and resource utilization is necessary.

Although hardware tools were popular in the 60's and early 70's (2,7,18,20), commercial versions are almost extinct for
uniprocessors [1] and do not exist for multiprocessors. Some degree of uniprocessor performance measurement can be accomplished with hardware instruments such as logic analyzers and in-circuit emulators [19], but their ability to provide useful trace measurement information is limited and they are not currently applicable to multiprocessors. A number of hybrid tools have been designed recently, some of which are applicable to multiprocessor architectures [9,11,12,15,16,17,22].

Our present focus is on multiple-instruction-stream, multiple-data-stream (MIMD), tightly coupled, shared memory multiprocessor systems. Based on our hybrid measurement experience and the fact that both non-intrusive trace measurements and resource utilization measurements are needed, a hardware tool that is capable of accomplishing these measurements has been designed. This paper discusses that design, the insight gained into its advantages and limitations for this class of multiprocessors, and the results of measurements made on two kernel (small benchmark) routines. This measurement tool is based on extensions to existing uniprocessor measurement techniques. Although designed independently, it is similar in concept to one designed by Gregoretti [10], but differs significantly in its implementation. As a result of developing this tool, we have realized that some technology trends limit its applicability and thus the ability to obtain a number of performance metrics.

**Instrumentation Design**

Events are defined by the state of certain signals. These states are called patterns and are user defined prior to an experiment. Dedicated hardware searches for these patterns, and, when they occur, causes samples to be taken. As these are signal states, they may be used to define not only trace information (e.g., program location), but also resource utilization (i.e., machine hardware activities).

The Resource Measurement System, REMS, consists of a set of Sample Units (SU) connected to an analysis computer via a bus, see Figure 1. An SU is the logic that captures, processes, and stores measurement samples. There is one SU for each CPU in the multiprocessor under test. An SU consists of three major modules, a Pattern Matcher, a Preprocessor and a Sample Memory.

The Pattern Matcher is the logic that constantly monitors the hardware signals of the processor and compares these signals against a set of stored signal patterns. When any of these patterns are matched, a trigger is generated that causes a measurement sample to be taken. The Preprocessor performs a data reduction on resource utilization events, e.g., cache hit ratio and bus utilization. In general, the Pattern Matcher is concerned with obtaining trace measurement information and the Preprocessor is concerned with obtaining resource utilization information. The Preprocessor and the Pattern Matcher of any single SU have separate and independent connections to the different signals they monitor and, therefore, do not have to be attached to the same processor. The Sample Memory is a large local memory which stores every measurement sample. A sample from a single SU consists of the combined data output of the Pattern Matcher and the Preprocessor.

**Probes**

Each Pattern Matcher is connected to the hardware signals of a processor via a set of probes which reduce electrical loading on the signals to which they are attached and extract a sample clock and state information from various timing signals. The sample clock is used to strobe a signal pattern (Trigger Data) into and through the Pattern Matcher. Each time the information on the CPU bus changes, the probes generate a new sample clock to capture the new Trigger Data. Each Preprocessor is connected to different signals via a separate but functionally similar set of probes.

**Pattern Matcher**

The Pattern Matcher, using the sample clock, latches the Trigger Data and compares it against a set of stored patterns. If the latched Trigger Data matches bit-by-bit any of the stored patterns then, a global trigger is generated that causes every SU to take a sample which provides a global view of activities. The output from the Pattern Matcher, which is part of the sample, is the Trigger Data.

Two design issues of the Pattern Matcher were its operational speed and the ability to match an arbitrary set of 32-bit patterns which may be changed for each experiment. An associative memory with a mask per entry was needed. This is not commercially available in the size required and we felt a custom designed one would be too expensive. Gregoretti [10] used a custom designed associative memory in the implementation of his OBSERVER module (functionally similar to our SU), but it was limited to only 16 patterns of 16 bits per pattern. Both Gregoretti's approach and our approach allow arbitrary don't cares to exist in the patterns. Our approach was a direct memory lookup technique utilizing a random access memory (RAM) as a Pattern Matcher where the trigger pattern is the RAM address and the contents of each memory location is a match/no-match indicator. The potential trigger pattern set is very small (hundreds) compared to the universal set (2**32, billions), and can therefore be recoded into a smaller state space, thus employing a smaller, more feasible, RAM. A mapping between the original trigger pattern set and the reduced, recoded pattern set can be accomplished in parallel on disjoint pieces of the trigger pattern. This is similar to the decomposition of a single-stage
combinatorial logic sum-of-products expression into a multi-stage sum-of-products expression.

Figure 2 shows a block diagram of the Pattern Matcher which contains two Trigger Data latches and three levels of RAM, one level for each sum-of-products stage. The first two levels of RAM handle the mapping into the reduced state space. The third level does the pattern match on this reduced state space. The output from one of the LEVEL 3 RAMs is the “Global Trigger”, which causes all the SU’s to take a sample. The output of the other LEVEL 3 RAMs represent “Selective Triggers” which are control signals for the Preprocessors.

The Pattern Matcher, implemented with easily available 45 nsec RAMs, must acquire samples at a 100 nsec data rate. The data, therefore, is buffered between LEVEL 2 and LEVEL 3 RAMs. For synchronization, a similar buffer is used in the Trigger Data path.

Pattern Generation

Each of the SU’s is mapped into a 256K byte address block within the VME address space of the analysis computer. The analysis computer loads a new pattern set into a Pattern Matcher via memory writes to the assigned address block.

The pattern set represents events on which samples are to be taken. Each position of a pattern may be true, false or don’t care. The user is required to formulate a set of 32-bit patterns which is the input to a pattern assembler program. This program does the reduction by recoding the set and then loads those codes into the RAMs of the Pattern Matcher. The pattern assembler program eliminates state space overlaps between patterns, although subsets are acceptable. Patterns will overlap only when don’t cares occur. Pattern that overlaps are decomposed into a subset pattern and one or more disjoint patterns by expanding the offending don’t cares.

Preprocessor

The predominant function of a Preprocessor is to acquire resource utilization information. Some examples of resource utilization metrics are cache hit ratios, access latencies, and duty cycles; all of which are accumulated counts. The Pattern Matcher could be programmed to trigger on each occurrence of these events, but the large quantity of data would rapidly flood the sample memory. Thus the Preprocessors, by accumulating these counts, relieves the Pattern Matcher from filling the sample memory with large volumes of unprocessed data.

We currently have two classes of Preprocessors. One type accumulates a single independent count whose value must be maintained and reported in full precision. An example of this type of Preprocessor is a timestamp, which accumulates counts of time ticks in 32 bit precision and reports the current full 32 bit value each time a trigger occurs. The current clock rate is 4 MHz.

The other type of Preprocessor counts pairs of related events whose values must be maintained in full precision, but need not be reported in full precision. This type of Preprocessor is a ratio counter, which accumulates a pair of counts, for example, cache hit ratio. In this example, one of the counters accumulates memory reads, while the other accumulates cache hits. Both counters accumulate 32-bit-precision counts. Only a single value, the ratio, is of interest and its accuracy does not require the full 32 bit precision. The ratio is reported as two floating point numbers with truncated mantissas and a common exponent. The division is done during postprocessing by the analysis programs. The total number of bits required to represent this pair is dependent on the accuracy required of the ratio. If a 1.5% error is acceptable, a 64 bit counter pair can be truncated to a total of 16 bits (6 bit numerator, 5 bit denominator, and 5 bit exponent), thereby allowing two independent counter pairs to be reported for each 32 bit Preprocessor sample output. A 0.1% error can be achieved by truncating a 64 bit counter pair to a total of 32 bits (14 bit numerator, 13 bit denominator, and 5 bit exponent), resulting in a single counter pair reported for each 32 bit Preprocessor sample output.

In order to correlate resource utilization with segments of the test program, the Selective Triggers, generated by the Pattern Matchers, are connected to individual Preprocessors through switches that are set prior to each experiment. Like the Global Trigger from each SU that are “wire ORed” together, the corresponding Selective Triggers from each SU are similarly “wire ORed”. Thus a Selective Trigger occurring at any SU will cause the counters of all of the Preprocessors attached to that Selective Trigger to be reset immediately after a sample, if any, is taken. Therefore, measurement can be taken over a specific segment of the test program if a Selective Trigger is generated to reset the Preprocessor counters at the beginning of each segment.

Sample Memory

Each 64 bit SU sample, which consists of the output from the Pattern Matcher and the Preprocessor, is stored in a 2M byte, dual ported Sample Memory. One port is used for inputting samples from the SU while the other port is connected to the analysis...
Virtual address, physical addresses of a program is not known prior to run time and in many environments is dynamic and may change during execution. Some of the newer VLSI microprocessor IC's have the memory management unit and/or both data and instruction caches designed onto the chip. Some examples of this trend include National Semiconductor's 32532, Motorola's 68030 and Intel's 80386. This means that neither the virtual address nor the cache "hit" and "miss" signals are available for triggering or counting purposes. Even worse, a number of accesses (instructions and data) may occur that are satisfied by the on-board caches and will never be observed by the hardware tool.

Also, a convenient means to connect to the signals is necessary, such as a plug or connector. Without such a connector, measurement is limited to highly knowledgeable and skilled individuals who have access to the internal design of the machine which might involve proprietary information. On our current testbed multiprocessor all of the processors' signal lines are available at a connector on each CPU board. The REMS probes are designed with a matching plug which makes the physical and electrical connectivity a relatively simple matter. Only a few other signal lines are required, to which "flying" leads are attached. This connector is not available on other manufacturers' CPU boards and on newer versions of the current testbed. Without this connector attaching to the required signals is impractical. This connectivity problem could be alleviated by measurement consideration during the design phase of the processor.

Architectural Problems

Architectural problems have to do with the way the architecture functions. Due to current VLSI circuit technology, it is virtually impossible to provide access to every logic signal of a computer. So there will be operations internal to CPU IC's that affect the accuracy of measurement and yet will never be observable. In some cases the operation is observable, but it is impossible under normal performance circumstances to distinguish between a number of possible source processes, resulting in inaccuracy of measurement.

Internal Operations

Similar to the signal access problem, in which signals internal to an IC are not available, hidden internal operations can erroneously indicate the occurrence of an event. Look-ahead and pipelining are used on the instruction stream in order to keep the instruction queue full. Instruction prefetch is used to implement this look-ahead. In some cases, data prefetch is used for similar reasons. This could cause inaccurate measurement in two ways. First, by detecting the address of an instruction or of data during prefetch, the time that will be sampled and recorded will be the time of the prefetch and not the time the event actually occurs. The time skew may not be significant unless very fine grain measurements are attempted. Second, since prefetch is only a statistical anticipation of potential use and not an assured use, there is always the possibility that, although fetched, the instruction or data will not be used. But the measurement tool will erroneously log the fetch operation as if the event had occurred. The error introduced by this false trigger is much more serious than the introduction of a slight time skew as in the first error type.

There is no feasible internal redesign that will alleviate this problem. A possible compromise would be to modify REMS into a hybrid tool and allow software triggering similar to that used by the hybrid TRAMS [15,16] measurement tool.

Process Distinction

When acquiring trace measurement information, virtual addresses are the predominant signals on which to trigger. When multiple processes are executing, it is impossible to determine which process accessed the virtual address that caused the trigger. This is bad enough if the only processes
executing on the machine are part of the program under test, but will give totally erroneous results if there are other processes executing that are unrelated to the program under test.

A possible solution involves modification to the REMS and to the scheduler within the operating system. A system call to the operating system would be inserted into each process to be measured which would be a signal to the scheduler. Then, each time a process is scheduled for execution, the scheduler controls a dedicated signal line which would cause REMS to turn on and off all pertinent circuits, or the scheduler would write the process number of the process to be measured (else zero) to a dedicated memory location. The latter is similar to the approaches used in the EGPA [9,11] experimental pyramid multiprocessor and by Hughes [12] in the Diamond analyzer. Both of these approaches would require modifications to the Pattern Matcher and the Preprocessor so they would become inactive when the scheduler output is zero. Again, both these solutions cause REMS to become a hybrid tool, since some program perturbation occurs due to the embedded code in the operating system.

**Physical Size Limitations**

As the number of CPUs under test increases, the number of SUs also increases thereby expanding the physical size of REMS and the enclosure necessary to house it. The current SU implementation consists of three circuit boards, each plugging into a VME slot in a card cage. A single card cage can house six SUs of the current implementation. An increase of one order of magnitude in the multiprocessor size, from 6 processors to 60, would require 10 card cages to house the REMS. As should be evident, the size of the measurement tool can very quickly dwarf the processor to be measured.

**Single Timestamp**

The use of a single timestamp among multiple SUs requires that when any SU takes a sample, all SUs must take a sample. This way correlation between samples is maintained by physical position in the sample memory. Thus the i-th sample entry in each memory is interpreted as occurring coincident with the single timestamp associated with that entry. An alternative and more costly approach would be to have separate timestamps for each SU resulting in increased circuitry and a 50% increase in the data acquired per sample.

With this single timestamp design a synchronization problem has been created. In our testbed multiprocessor the sample clock is not periodic, the time between ticks can vary from 100 nsec to several microseconds. Also the sample clocks at each SU are asynchronous to each other. If a single SU has a burst of triggers closely spaced in time or if multiple SUs triggers closely spaced in time, then it is possible that for some SUs multiple triggers will arrive within the same sample clock period causing only a single sample to be taken; while for other SUs they will arrive during different sample clock periods causing multiple samples to be taken. The result is a misalignment between samples and the timestamp in the sample memory.

**Performance Measurement**

We present two programs which were used to measure the performance of various architectural features of a tightly coupled, shared memory multiprocessor. The first program is a kernel test code used to exercise memory access, by continually copying one array to another, where the copy size is variable. The second test program is a parallel implementation of the sieve of Eratosthenes presented by Bokhari [3] which is an algorithm to compute prime numbers.

**Input Data**

The 39 bit input data for the pattern assembler program is formatted in binary as 2 fields: input patterns and output triggers. The input pattern field is further divided into 3 sub-fields: a 4 bit qualifier, 4 bit user defined, and 24 bit cpu bus data (virtual/physical address or data). The output trigger field indicates which triggers, global and selective, will be active for a match of this input pattern. Figure 5a shows the input data for the pattern assembler program taken from the first test program. An "x" in any bit position of the input pattern corresponds to a "don't care".

The two leftmost bits of the qualifier denote the cpu bus data type as follows:

- 00 = not valid
- 01 = virtual address
- 10 = physical address
- 11 = data

The next to the rightmost qualifier bit denotes the user/kernel mode for the current sample, while the rightmost bit denotes mode for the previous clock time. Thus, not only is user/kernel mode known, but it is easy to detect the transition from one to the other. A "0" in the pattern indicates kernel and a "1" indicates user mode. The virtual addresses, as shown in the comments, were obtained from the symbol table of the compiled program. The "x" preceding the address indicates hexadecimal representation, otherwise decimal.
preprocessors in the scale factor. If between them. Each set has the capacity of two virtual pages, which holds both data and instructions without differentiating least-recently-used sample number, Memories) configured with two trigger the 2 cache hit ratio. If the implied sixth denominator bit is one and the scale factor also requires a correction of minus one, see Figure 5c. Memorv Kernel Results

The testbed multiprocessor has a 2-way set associative cache (block) is 8 bytes wide. Thus a page of data will straddle 2 virtual pages. The data sets for our experiment used five different copy sizes:

- 4K (128 bytes)
- 8K (256 bytes)
- 16K (512 bytes)
- 32K (1024 bytes)
- 64K (2048 bytes)

Each data set was run four times, once on 1 CPU with the other CPUs idle (except for the operating system), a second time simultaneously on 6 CPUs, a third time simultaneously on 10 CPUs, and a forth time simultaneously on all 16 CPUs.

The cache operates on physical addresses (PAs), and there is no correlation between virtual addresses (VAS) and PAs prior to actual execution. Therefore, depending on the run time PA allocation of the data, the resulting cache allocation can be heavily skewed. For example, the cache can hold 4 virtual pages of data. If we used a buffer size equal to 4 virtual pages one may think this would exactly fill the cache and result in no cache misses. But depending on the VA to PA mapping, all 4 pages could map into the upper half of the cache resulting in continuous cache misses. Figure 7a shows the possible range of cache misses as a function of the buffer size (at or below 1K there are no expected cache misses). The measured misses, for a given cache skew, always equaled or exceeded these values by as much as 200. These additional misses are due to interrupts and instructions that, depending on their PA allocations, cause data in the cache to be overwritten which would otherwise stay in the cache.

The executable code of the inner loop of Figure 6 consists of 8 machine instructions, totaling 18 bytes packed into 5 machine words. Only two memory references exist within this loop, a read from buffer_A and a write to buffer_B, all other data are immediate or in a processor register. The first experiment was to sample the actual VA stream, which is a mix of instruction prefetches and data references, and compare it to the executable code. The actual VA stream of this inner loop consisted of 4 instruction prefetches (4 bytes each), a data read, an instruction prefetch, a data write, and another instruction prefetch. Due to the LRU cache replacement policy these 6 prefetched instruction words will always be in cache. Thus 6 instruction words (24 bytes) are prefetched when only 5 (18 bytes) are required, resulting in an access overhead of 1/6 = 16.7%, or on a byte level 6/24 = 25%. This inner loop executed in 6.0 to 6.25 micro seconds (our time resolution is 250 micros) when no MMU or cache misses occur, and consists of 7 reads (6 instruction prefetches and 1 data read). This was measured using our smallest data set. A prefetch takes about .75 micro which is a time overhead of 12%. A cache miss will only occur for the data read, yielding a minimum cache hit ratio of 6/7 = 86%. But since the replacement policy fetches a block of 2 words, a cache miss will occur at most every other loop iteration yielding a minimum cache hit ratio of 13/14 = 93%.

Our actual measured value of 92% is slightly lower due to interrupts and truncation errors.

![Figure 6. Psuedo code of the memory test kernel program.](image)

It exercises the access between the processor and the memory, both local and global.

Output Data

The raw output of the REMS (the contents of the Sample Memories) configured with two trigger boards and two preprocessors is shown in Figure 5b. The first column is the sample number, the second and third columns each are a trigger data/preprocessor data pair. The time preprocessor is a cumulative timestamp in 250 nsec increments.

The ratio preprocessor data represents two independent counter pairs, where each pair consists of two 32 bit counts condensed into 16 bits. Each truncated 16 bit representation is formatted as a 5 bit scale factor (a common exponent), a 5 bit denominator, and a 6 bit numerator. The value of an implied sixth (high order) bit in the denominator can be determined from the scale factor. If the scale factor is zero, then the implied sixth denominator bit is also zero. If the scale factor is nonzero, then the implied sixth denominator bit is one and the scale factor requires a correction of minus one, see Figure 5c.

Memory Kernel Results

This program continually copies one array to another; the copy size is variable. It is expected that if the copy size is small all reads are satisfied by the cache, but all writes access shared memory (due to a cache write-thru policy). This results in a 100% cache hit ratio. If the size is large all (data) reads and writes must access shared memory, resulting in a cache hit ratio less than 100%. Figure 6 is the pseudocode of this program.

The testbed multiprocessor has a 2-way set associative cache which holds both data and instructions without differentiating between them. Each set has the capacity of four virtual pages. A least-recently-used (LRU) replacement policy is applied between the 2 sets. The memory access is 4 bytes wide and a cache line (block) is 8 bytes wide.

The size of data buffers A and B of Figure 6 were statically allocated at compile time, but the copy size of the buffers was specified at run time. The buffers do not start on a page boundary, thus a page of data will straddle 2 virtual pages. The data sets for our experiment used five different copy sizes:

- 32 x 4 = 128 bytes (fits a fraction of a page, and does not collide with the instructions)
- 512 x 4 = 2K bytes (1 page, but straddles 2 pages)
- 1024 x 4 = 4K bytes (2 pages, but straddles 3 pages)
- 2048 x 4 = 8K bytes (4 pages, but straddles 5 pages)
- 3072 x 4 = 12K bytes (6 pages, but straddles 7 pages)
- 4096 x 4 = 16K bytes (8 pages, but straddles 9 pages)

![Figure 7a. Range of Cache Misses vs Copy Size](image)
When a cache miss does occur an additional 75 μsec to 1.00 μsec is required to execute that loop iteration. This is an overhead of 12% to 17% for a single loop, but since it usually occurs every other iteration the total overhead is 6% to 9%.

The only interrupt that occurs is due to the system clock at regular 10 msec intervals (± 5 μsec) and causes a short interrupt of 200 μsec to 300 μsec in duration. Although periodically a long interrupt occurs with a duration of 500 μsec to 700 μsec. Overall this works out to be approximately a 3.5% overhead.

The total overhead encountered leaves:

100.0%  
-3.5% (interrupts)  
-12.0% (unused prefetches)  
-7.5% (cache miss)

77.0% useful work.

Thus 23% of execution time of this test kernel was overhead time. This would not normally show up using standard software measurement tools. In fact delays due to cache misses and unused prefetches would be counted as useful user time. Figure 7b shows the bus utilization characteristics for this experiment. Even though bus utilization was measured as high as 83%, the execution times did not vary by more than 6%.

This type of behavior can be generalized to algorithms which have short loops, such as matrix multiplication and data copying, and have some sequentiality of data. Thus, on this type of architecture, the number of instructions prefetched will dominate the cache hit ratio of tight inner loops, yielding misleading cache hit ratios. The impact of the time used to satisfy cache misses and unused prefetched instructions are difficult to determine since it is dependent on the internal parallelism of the architecture as to whether or not other activities are blocked.

### Sieve Kernel Results

This test program is a parallel implementation of the Sieve of Eratosthenes presented by Bokhari [3] which is an algorithm to compute prime numbers. The concept is to have a controller (a parent process) which sends runners (child processes) down the remainder of a number list marking multiples of their starting number. Upon completion all unmarked numbers in the list are primes. Figure 8 shows the pseudo code description of the heart of the control process and the runner processes. The number list, which can be very long, must be kept in shared memory so that both the controller and all the runners can access it. A communication/synchronization variable for each runner, "starting_number[runner]", also must be in shared memory.

The Sieve algorithm was run with several values for both of the input parameters. One of the parameters specifies the length of the number list. Values of 10K, 100K, 1M, and 10M were used. Paging was not a factor since main memory was large enough to hold the entire list. The other parameter specifies the number of runners available, where each runner has a dedicated processor. Values of 0 to 15 were used. The case of 0 represents a uniprocessor where a single process does both the control and the running in a serial order. For all the other cases there is a dedicated control process and from 1 to 15 runners. The control process and the runners all execute on dedicated processors. For measurement consistency the input parameters were entered through a file rather than manually via a terminal.

In order to evaluate the sieve algorithm on this architecture we start at the highest level of the measurement hierarchy, overall...
response time. Figure 9 shows the elapsed wall clock time as a function of runners and list size. For each list size there is an optimal number of runners to use, as is indicated by the minimum of each curve in Figure 9. To probe further we selected various segments of code and acquired their individual execution times. The segments selected correspond to the individual pseudo code statements of Figure 8.

The runner process is fairly simple, it locks itself onto a processor, initializes its portion of the list, then it interleaves between waiting and executing. It busy waits on a shared variable for an execute command from the control process. Busy waiting does not add to the bus load. The shared variable is cached when first accessed and additional read accesses (busy waiting) are to the cache until the variable value is changed, at which point the bus watcher circuitry invalidates the cache copy, forcing a memory access. The lock time is consistently between 1800 μsec and 1850 μsec. The runner initialization time decreased inversely proportional to the number of runners for a given list size, as expected. The amount of initialization time required by the runner processes were about the same, but differed from the initialization time required by the control process, which took about 30% more user time. The wait and execute segment times varied widely and are discussed further, below.

The control process gets the input parameters, spawns the specified number of runners, initializes its portion of the list, waits for all the runners to complete and then begins the control process algorithm. When the control process completes, it waits for all of the runners to complete and then terminates the sieve algorithm. The parameter input times were consistent between 41 msec to 43 msec. The variation was due to the difference in the number of characters in the input file. The initialization time of the list decreased inversely proportional to the number of runners for a given list size, similar to that of the runner processes, but comparatively took 30% longer. The spawning measurement was taken from both the parent and child perspective. The starting point for both was the beginning of the spawning loop in the parent process. The end point for the parent was the beginning of the next iteration of the same spawning loop. The end point for the child was its first executable source code instruction. The spawn times measured from the parent process were found to be sensitive to the overall program size, but independent of the number of children spawned. As the list size grew the spawn times also grew, as shown in Figure 10. The spawn times measured from the runner processes were similarly sensitive to overall program size, but in addition exhibited a bimodal behavior, either taking about as long as or twice as long as the parent process. The time required to spawn successive child processes, from the child’s viewpoint, did, in most cases, interleave between these two values. This behavior was traced to our implementation (requiring an accurate measurement) of locking processes onto processors. Since the parent process was locked onto a given processor, each child it spawned inherited that same processor and had to wait until the parent process was not using the processor before that child could switch to its own dedicated processor. When processes were not locked to processors, this behavior did not occur, and the spawn time was the same for parent and child, since the scheduler would automatically move a newly spawned child to a free processor.

The synchronization wait after list initialization is due to the variation in spawning times. The work to initialize the list is equally distributed between processes. When the control process finishes its initialization of the list, all but the last one or two runners spawned, have already initialized their portion of the list and set their synchronization flag. Thus their synchronization is immediate, requiring only 16 μsec for a single pass through the test loop. Depending on the spawn times for these last few runners, they may complete their list initialization before or after the control process. When these last few runners complete after the control process, the control process busy waits on their synchronization flag, thus creating the observed variation in synchronization wait times.

The control process startup overhead consists of a parameter input segment whose execution time is a constant, a spawning segment whose execution time increases linearly with the number of runners and also increases with the size of the list, an initialization segment whose execution time decreases inversely proportional to the number of runners and increases linearly with the list size, and a wait segment which is a random variable related to spawn time. Thus for a given list size, there is an optimal number of runners that can be effectively used during startup before the time required to spawn another runner is longer than the time saved in initializing the list.

The execution time of the control portion of the sieve algorithm decreases as the number of runners increase, and approaches an asymptotic value dependent on the list size, see Figure 11. The value of the asymptote grew linearly with the list size, within the range measured. Keeping the number of runners constant and increasing the list size results in an execution time that grows faster than the corresponding increase in list size, thus as the list size is increased more runners are necessary to achieve the asymptotic execution time. It takes about 18 μsec for the control process to test if the end of the list has been reached and if the current list position is unmarked, see Figure 8. Once an
unmarked position is located, it takes about 12 μsec to test if a runner is available. Once an available runner is found it takes about 8 μsec for the control process to dispatch it, and 20 μsec to 30 μsec for the runner to activate. By examining the execution times of the runners and the sequence in which they are dispatched, we notice that the runners complete in a LIFO order. This is because each successive runner is taking a larger stride through the list than the previous runner, and thus completes faster. Also the very first runner is the limiting factor. No matter how many runners are available, the algorithm cannot complete any faster than it takes the first runner to complete. This, therefore, explains the asymptotic behavior indicated in Figure 11.

During the control portion of the sieve algorithm the cache hit ratios are very close to 100%. This is mainly due to instruction fetches of the few small loops, and caching of synchronization variables, as seen in the memory kernel above. At the beginning of the control portion of the sieve algorithm system bus utilization jumps as each runner process accesses new code from shared memory for its exec loop, and then quickly settles to about half of this maximum for the remainder of the algorithm. Process termination was noted as causing a heavy load on the system bus. The measured bus utilization varied significantly during the execution of this algorithm, recording bursts of over 80% although even in these cases execution times did not vary by more than 2%. This only means that bus utilization is not necessarily a strong indicator of memory contention and processor delay. For this set of experiments these latter metrics were not measured, because on these distributed protocol buses somewhat complex circuitry is needed to determine when these events occur since request and response are separated by other similar events and overlap of operations occur. These measures will be pursued in the future.

Conclusions

The design of the Resource Measurement System (REMS) has been presented as an example of a non-intrusive hardware measurement approach for a shared-memory multiprocessor architecture. It is able to obtain both trace measurement and resource utilization information. It follows the design precepts of Franta [6] in that it does not interfere with the executing program and provides a global view of program activities. Of interest in the hardware design is the Pattern Matcher, which is able to simultaneously search through hundreds of patterns. In contrast, logic analyzers can only search for a single pattern at a time, and other hardware/hybrid experimental monitors [9,10,11,13] can only simultaneously search for sixteen or less.

Some of the limitations of using measurement tools like REMS on shared-memory MIMD architectures have been described, as have some potential solutions. These limitations may explain why hardware monitors for commercial (vs. experimental) computers are becoming extinct.

In general, the most realistic solutions to these limitations result in converting REMS into a hybrid tool by implementing a software triggering mechanism in addition to the existing hardware pattern matching. For our tested multiprocessor we were able to avoid many of these limitations because: (1) signal accessibility was provided by an extra CPU socket on the processor board; (2) architectural problems were avoided since the cache and the MMU are both external to the CPU chip; (3) process distinction was overcome by locking processes onto CPUs. Since the prefetch queue on our tested multiprocessor is flushed at any branch (forward or backward), we chose, when possible, virtual addresses that are at the beginning of a loop to catch the actual instruction fetch. Erroneous prefetches still occurred as code prior to the loop was executed.

We feel the REMS configuration as presented in this paper is too expensive a tool for a shared-memory multiprocessor environment, especially when the limitations discussed are considered. A more feasible and cost effective configuration might be a single Pattern Matcher with a timestamp Preprocessor on the system bus along with a number of ratio-counter Preprocessors. This configuration would be used as a hybrid measurement tool rather than a hardware tool. Code to trigger samples would be inserted into the program to be measured. The added code would write to a reserved set of physical addresses which would be mapped into the virtual address space of the program. The single Pattern Matcher upon matching a pattern to a physical address would trigger the taking of a sample consisting of that address, the timestamp, and the contents of all the other ratio-counter Preprocessors. Thus, during analysis the cause of the sample can be determined by the specific physical address stored in the sample. We have not planned an entire measurement environment as discussed by Segall [23], although we have plans to expand the user/language interface tools as discussed by Burkhart [4], in an effort to provide measurement facilities to the user as part of the normal compilation and execution environment.

We feel that the Pattern Matcher and the ratio-counter Preprocessor are interesting and useful items by themselves. The Pattern Matcher effects an associative match on a relatively large number of patterns, where each pattern contains the equivalent of a separate don't care mask. This could provide an alternative to associative memory which, for a size large enough for many applications, is not commercially available and is expensive to build. The ratio-counter Preprocessor provides a means of condensing 64 bits of counter data into 16 bits or 32 bits depending on the accuracy of the desired result. This can be important in VLSI situations where pins are a limitation and also in data storage situations where volume is a limitation.

With the current interest in multiprocessors, performance information has become a major concern to many users. The focus of such measurement is to ultimately increase performance. It may be argued that resource utilization information is useless or irrelevant to users of commercial multiprocessors since they can’t change the embedded hardware and it’s the cost effectiveness of this VLSI embedded hardware that counts. We feel that this class of user needs tools to evaluate their algorithms on various architectures. They select the "best" architecture for their algorithms and in addition further tune those algorithms for that architecture. These evaluations require more than just "macro" trace measurement information, such as execution time. They also require "micro" measurement to determine the underlying execution characteristics which includes resource utilization information obtained in such a way so as not to perturb the actual execution characteristics of their code.
Acknowledgement

The Authors wish to thank R. J. Carpenter for his extensive editorial comments in the preparation of this paper and for raising some of the issues discussed herein.

References
