The SHRIMP Performance Monitor: Design and Applications

Margaret Martonosi martonosi@ee.princeton.edu
Douglas W. Clark doug@cs.princeton.edu
Malena Mesarina mesarina@cs.princeton.edu

Departments of Electrical Engineering and Computer Science
Princeton University

Abstract

Growing complexity in many current computers makes performance evaluation and characterization both increasingly difficult and increasingly important. For parallel systems, performance characterizations can be especially difficult to obtain, since the hardware is more complex, and the simulation time can be prohibitive. The challenge is to design a low-cost and yet flexible and powerful performance monitoring system to provide systems implementers and application programmers with detailed performance information.

This paper describes a performance monitoring system for the SHRIMP multicomputer. The system's core is a hardware monitor with several novel features including multi-dimensional histograms, page tags, histogram categories, and a threshold interrupt mechanism. We also describe software applications that make use of these features. These applications range from fairly simple code-oriented or data-oriented performance tools, to more complicated on-the-fly use of the monitor to improve the performance of a shared virtual memory system. We have found that the concurrent development of the hardware and software portions of the system has led to a novel design that supports a wide range of hardware and software uses.

1 Introduction

Over the past decade, parallel computers have come into much more widespread use. In particular, a current focus of research in the parallel community has been on "convergence machines" [13], which are created by using high-performance networks and specially-designed network interfaces to interconnect commodity PCs or workstations [3, 16, 24]. In convergence machines, the design effort is often focused on supporting fast data access and communication amongst an application's co-operating processes.

This paper appeared in the *1996 ACM SIGMETRICS Symposium on Parallel and Distributed Tools.*

Initial design decisions in such efforts are generally made with the help of analytic methods and simulation results. Many deeper questions about the detailed behavior of the final system cannot, however, be answered through simulation, because it is too slow to use on significant applications. Furthermore, it is often impossible to collect simulation-based results that fully reflect the effects of multiple processes and the operating system. Support for monitoring the running system is needed.

Monitoring methods based entirely on software are valuable in some cases. Some performance information (such as instruction counts or explicit synchronization delays) may be both sufficiently coarse-grained and software-visible that it can be monitored simply by instrumenting software or by using techniques like program counter sampling [11]. Many current parallel machines, however, allow implicit communication that is hard to monitor without hardware support. In cache coherent, shared-address-space machines, much of the interprocessor communication occurs via the hardware cache-coherence mechanism. With communication potentially occurring on any read or write, monitoring must be very fine-grained. With the latency of such communication dependent on the dynamic interleaving of accesses from different processors, it is virtually impossible to measure in software, except via simulation. Even in message passing machines, increased support for implicit, fine-grained communication has made monitoring difficult.

For these reasons, hardware performance monitors are an attractive approach for measuring a running system's behavior. These measurements can be used to modify mutable parts of the system hardware, to help with software development and tuning, to thoroughly characterize and evaluate the behavior and performance of a prototype system, and to guide the design decisions of future efforts. Furthermore, software can also use performance information to make on-the-fly decisions about how to adapt to observed behavior. (For example, a parallel application might migrate a page of data to the processor that references it most often, based on performance numbers available during execution.)

This paper describes the design and uses of the hardware performance monitor for the Princeton SHRIMP multicomputer [3]. The main focus of the SHRIMP project has been on designing hardware and software support for low-cost, user-level communication mechanisms. Thus, we have designed hardware and software to monitor system behavior, with a particular emphasis on monitoring the inter-processor communication statistics. In designing the monitoring system we aimed to create a simple, yet flexible and powerful hardware monitor and seamlessly integrate it with the higher-level software layers that take advantage of it. Thus, the hardware and software in our performance monitoring system have two primary goals: effective characterization of system behavior, and the support of on-the-fly monitoring by software. The research contributions of this work are twofold. First, we have designed hardware that simultaneously supports both general system characterization and on-the-fly software adaptation. Second, we have identified and evaluated some of the software applications that can make use of this system.

Section 2 gives background information on the SHRIMP multicomputer. Section 3 describes specific features of the performance monitoring hardware design, as well as design alternatives and tradeoffs. Following this, Section 4 presents several examples of using the performance monitor for fine-grained software support and gives preliminary evaluations. We discuss related work in Section 5 and give conclusions in Section 6.

2 SHRIMP: An Overview

The SHRIMP (Scalable High-performance Really Inexpensive Multi-Processor) project at Princeton studies how to provide high-performance communication mechanisms in order to integrate commodity desktop computers such as PCs and workstations into inexpensive, high-performance multicomputers [2, 3]. SHRIMP nodes are unmodified Pentium PC systems, each configured with disk and standard I/O devices such as tape drives, monitors, keyboards and LAN adaptors. The network is the Intel Paragon mesh routing backplane [26]. The connection between a network interface and the routing backplane is via a simple signal-conditioning card and a cable.

Figure 1 shows a SHRIMP multicomputer prototype system. The highlighted components in the figure correspond to the experimental system components being designed and implemented at Princeton. The right hand side of the figure focuses in on a single SHRIMP compute node, a standard PC system.

A main goal of SHRIMP is to provide a low-latency, high-bandwidth communication mechanism whose performance is competitive with or better than those used in specially designed multicomputers. The SHRIMP network interface implements virtual memory-mapped communication to support protected, user-level message passing, and fine-grained remote updates and synchronization for shared virtual memory systems. SHRIMP supports a variety of programming styles by supporting communication through either *deliberate update* or *automatic update*. With deliberate update, a processor sends out data using an explicit, user-level message-send command. With automatic update, a sending process can map memory within its address space as a send buffer; any time the sending process writes to one of these mapped (outgoing) memory regions, the writes are propagated automatically to the virtual memory of the destination process to which it is mapped.

It is in large part this implicit, automatic-update communication that spurred our monitor design. Without hardware monitoring support, one would need to instrument (i) all writes in the code in order to identify which resulted in communication, and (ii) all reads in the code, to see which reads received “old values” and which got “new” values, and therefore estimate the message latency. The monitoring hardware makes it possible to observe events (such as communication on individual reads and writes) with lower overhead than software techniques; more importantly though, it lets us observe events (such as automatic-update message arrivals) that are not always observable at the software level.

3 Hardware Monitor Design

A performance monitoring board is located at each SHRIMP node, and captures information at the arrival of incoming packets. Figure 2 shows a block diagram of the performance monitor. Given the monitor’s connectivity, we can observe events on both the network interface and the EISA bus. The discussion focuses on network interface behavior, however, since that is where interesting application communication behavior will be visible.

The monitor responds to user commands (e.g. start, stop, etc.) encoded as special EISA bus writes. (EISA writes are also used to configure the board by reading or writing its registers, but those connections are not shown in the figure.) Once the monitoring has begun, the network interface sends the monitor a copy of each raw packet as it is received, and the monitor parses the raw packet data to extract the fields of interest. It then updates its statistics memory appropriately, checks if any new EISA commands have been issued, and waits for the next packet. Like some previous performance monitors (e.g. [12]) we have a flexible hardware design based on FPGAs, but we have designed mechanisms into the monitor for *runtime* flexibility as well. The subsections below discuss some of the key features in more detail.

3.1 Histogram and Trace Modes

The monitor has two modes: *histogram mode* and *trace mode*. In histogram mode, it increments a count associated with a runtime-selectable set of packet characteristics; in trace mode it appends packet-specific information to a sequential trace in DRAM. A user-writable register indicates whether the system should operate in histogram or trace mode. As described in Section 3.4, we also provide a mechanism for switching dynamically from histogram to trace mode while data is being collected.

In histogram mode, variables of interest to the experimenter are used to form the address lines of the bank of DRAM (labeled “histogram and trace memory” in Figure 2). On each event, the memory contents so addressed are incremented and written back. At the end

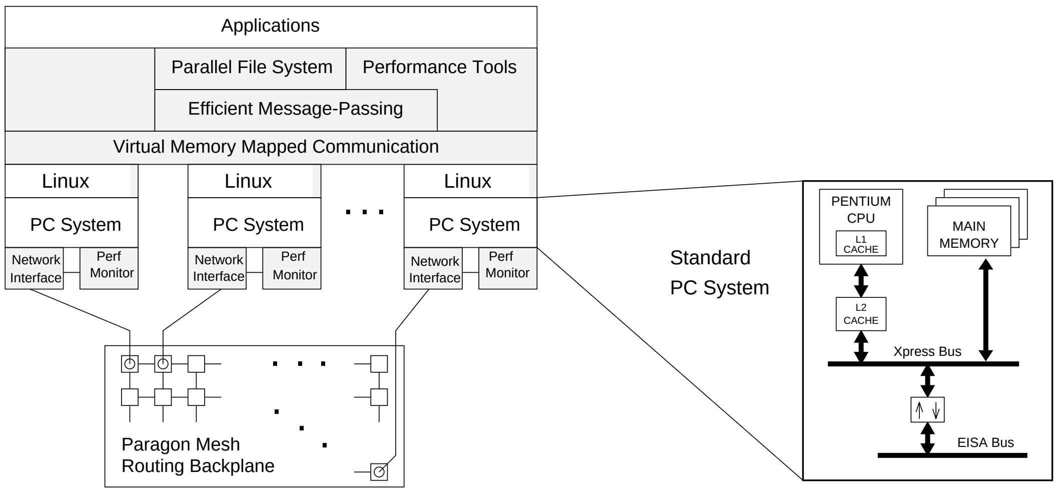

Figure 1: An overview of the components of the SHRIMP system.

of an experiment the memory contains a histogram of the various values taken on by the variables. By using more than one variable to form the address, the result is a "multi-dimensional" histogram showing the joint distribution of the variables. The design includes multiplexing to let users select the bit fields forming the address. This gives the monitor runtime flexibility to measure many different combinations of variables and show correlations between them.

Trace mode shares logic with histogram mode, but uses it in a complementary way. In trace mode, the memory bank is addressed using a counter that is incremented after every trace count. The concatenated bitfields (which are the *address* in histogram mode) are the *data* written to the trace memory location; they summarize interesting details about this trace event. (In trace mode, we also include 20 of the current time bits in each trace event.) For both modes, the monitor's design is facilitated by its relatively modest speed requirements: in the SHRIMP prototype the minimum inter-packet time is 625 nanoseconds, so that inexpensive dense DRAMs are quite suitable for the histogram/trace memory.

3.2 Measurable Variables

The monitor design lets experimenters select combinations of five possible variables for histogram construction, or for inclusion in the per-packet trace data. These metrics are:

- **Packet latency** Each SHRIMP network interface board and each performance monitor maintains a 44-bit global clock register. These are synchronously controlled across the whole system by 10 MHz clock signals distributed by the Paragon backplane.

  At the sending node, the network interface inserts its copy of the global clock into each outgoing packet as a timestamp. At the receiving node, the performance monitor reads its copy of the global clock as the packet arrives, and subtracts the packet's timestamp from it; this yields the packet's end-to-end hardware latency in 100-ns. cycles. Since the resulting difference will be a 44-bit quantity but our DRAM address is only 20 bits wide, we give users optional control over the latency's resolution. The default action is to consider only the lower 20 bits of latency, but at this point, one can adjust the latency's resolution by right-shifting it a user-selectable amount. Values greater than a user-specified maximum, or less than a user-specified minimum, force the latency to all 1's, or all 0's, respectively.

- **Packet size** As the remaining packet data streams in, a counter in the monitor keeps track of the packet's size in bytes. A 12-bit field is delivered to the address multiplexers.

- **Packet's sender** The 8-bit identity of the sending node is part of the packet header, and is sent to the address multiplexers.

- **Category** User-level software can set a 4-bit category register on the monitor board. As described in Section 4, this allows users to attribute statistics gathered by the hardware back to the responsible software constructs.

- **Page tag** The performance monitor has also been designed to provide support for coarse-grained data-oriented statistics, by allowing different pages in memory to be assigned different *page tags*, which can then be used in histogram statistics. The 4-bit tags are stored in SHRIMP's incoming page table entries on the network interface board; they are read out at the same time as the page-mapping information [3]. Updating page tags requires (protected) memory-mapped I/O writes of the appropriate locations in the page tables.

Once calculated, all five quantities are fed into the address multiplexer, which generates the memory address of the histogram bin that is to be incremented. This multiplexer is described next.

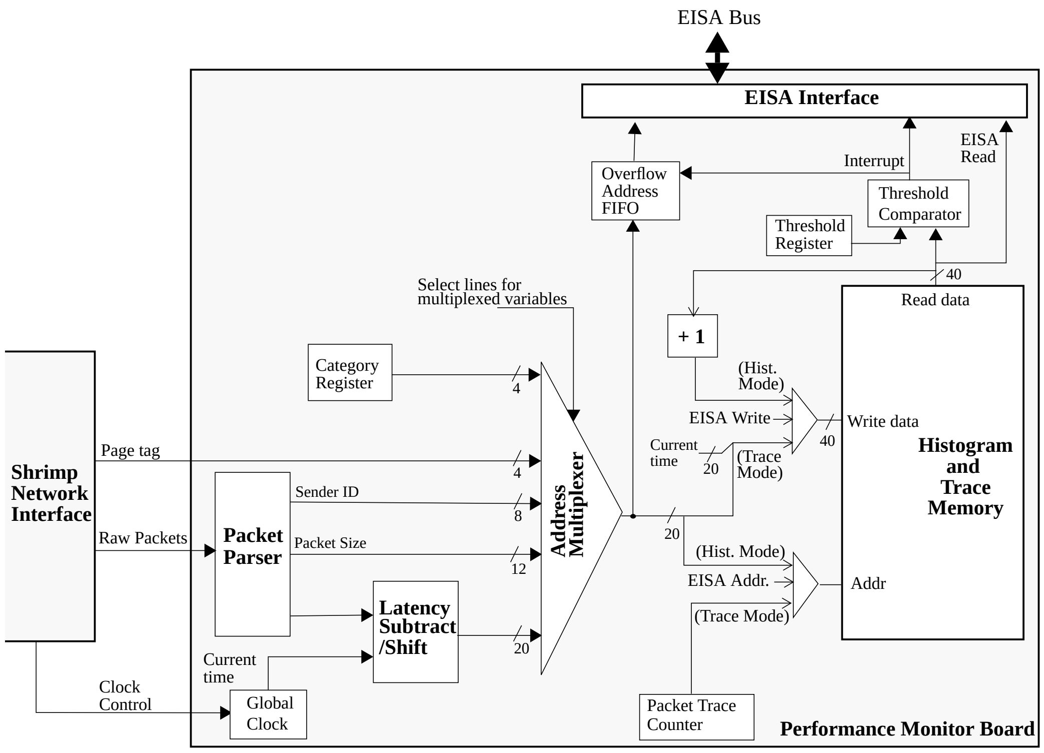

Figure 2: Hardware Performance Monitor block diagram.

3.3 Multiplexing Variables for Flexible Monitoring

Multi-dimensional histograms allow the experimenter to build up a two, three, four or even five-dimensional array of counters in the histogram memory. For example, one might want to collect histogram statistics using both packet size and packet sender, to yield the joint frequency distribution of these two variables. A correlation between the two might indicate that a particular SHRIMP node tends to send larger packets on average. We describe some more elaborate experiments in Section 4.

The histogram memory in our design contains one million 40-bit words, expandable to 16 million words. The 20-bit address can be composed from the five variables in a large number of ways. We divide the 20 bits of address into five fields, each four bits wide, that can be set independently. Figure 3 illustrates our approach. In our current design, latency is 20 bits wide, packet size is 12 bits, and sender ID can be an 8-bit quantity. The other two metrics, page tag and category, are 4-bit quantities. Thus, there are 12 different 4-bit chunks to be multiplexed into five possible positions in the address. This means there are $12!/(5! \cdot 7!)$ or 792 possible multiplexer setups.

The situation is simplified, however, by adding the restriction that latency fields must be contiguous. That is, we assume that users would never want to use the 4 high-order latency bits along with the 4 low-order bits, without using the intervening 12 bits. A final restriction is that the sender ID field must either be used in its entirety, or not at all. These restrictions drop the number of possible combinations to 190, all of which can be selected by the monitor.

More importantly, the two restrictions placed on address generation reduce the number of inputs to each of the five multiplexers. With these restrictions, multiplexers can be arranged so that none of them requires more than 8 inputs. Two of the multiplexers have 7 inputs. Since this logic is implemented (as is the bulk of the design) as part of a FPGA, we can later reprogram the FPGA to refine this multiplexing scheme if warranted. (We also have incorporated a number of jumper wires into the design so that additional high-order bits can be used to address extra memory, without recompiling the FPGA. These jumper wires can be connected to specific signals either on the performance monitor board or on SHRIMP's network interface board.)

3.4 Using the Threshold Register

For performance monitors to be useful in on-the-fly monitoring, they must be able to interrupt the CPU on certain "interesting" events. Without this ability, higher-level software would have to query the monitor periodically to determine its status. In the histogram-based monitor, a particular 'event' may involve a large memory region; for instance, detecting packets from a particular sender would mean checking all 2^{12} bins with that sender's 8-bit address. Thus, reading the histogram locations intermittently throughout the run of a paral-

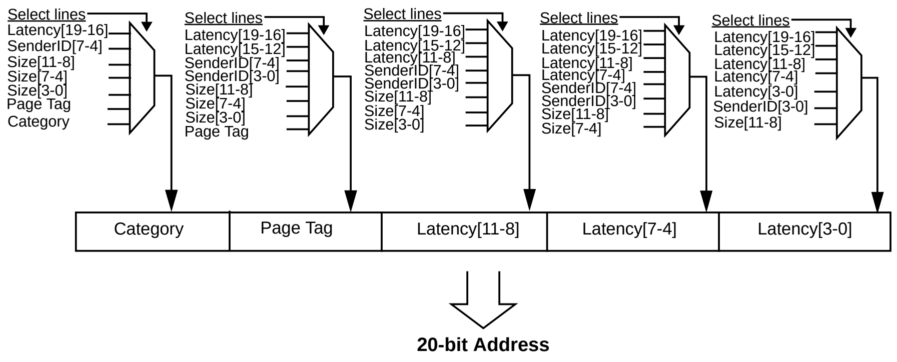

Figure 3: Multiplexing metrics to form a 20-bit histogram address. In this example, the *category* and *page tag* bitfields are combined with the low-order 12 bits of the latency to form a 20-bit histogram address.

lel application can often be both time-consuming and disruptive.

The SHRIMP monitor provides efficient support for selective notification. When running in histogram mode, the monitor has a threshold-interrupt feature that notifies the CPU when a count value passes a user-specified threshold. The threshold register is compared with the incremented value on each histogram update; if the value is too big, an interrupt is generated. The monitor also saves the histogram address that caused the interrupt. The software interrupt handler can read the overflow address register to see which bin caused the interrupt, in order to decide how to react.[1] This low-level mechanism can be used by higher-level application or operating system policies to take software action in response to monitored behavior.

The monitor can also be configured to use *triggered event tracing*. That is, when the monitor detects a threshold event, it not only signals an interrupt, but also can switch from histogram to trace mode automatically. This feature allows for long periods of histogram-based monitoring, followed by detailed tracing once a particular condition is detected. Triggered tracing and threshold-based interrupts can both be used in several interesting ways, including guarding against bin overflow, providing on-the-fly information to running software, or capturing error traces once an error condition is detected. Several applications are discussed in more detail in Section 4.

3.5 Monitor Control Software

Finally, the monitor responds to a set of low-level commands that come in via memory-mapped I/O writes on the EISA bus. These commands provide a basic library of efficient access routines on top of which higher-level tools can be built. The command set includes operations to start and stop monitoring, initialize ranges of histogram memory, and read ranges of histogram memory. In addition, one can adjust the resolution of the packet latency, switch the address multiplexer/shifter for the histogram address, and write the category and threshold registers. Because the monitor board's registers are memory-mapped, we can take advantage of SHRIMP's automatic update facility; once exported for remote use, a node's monitor board registers can even be written by *remote* nodes in the system.

The performance monitor's EISA Interface sees the commands as writes on the EISA bus, accepts the data, and acts on that data to update its control registers and read or write DRAM as needed. To make use of the monitor's ability to do fine-grained categorization of the statistics, we hope to make commands from the processor to the performance monitor have very low latency. When commands are generated by the local node, they incur the overhead of only a single EISA bus write. With no contention, the latency of this operation would be roughly 15 processor cycles on the prototype's 60 MHz Pentiums. For many monitoring uses (such as initialization and post-experiment reading of the histogram memory) this overhead is quite tolerable. To further reduce this overhead, we could take monitor commands from the PC's Xpress bus rather than from the EISA bus. This new interface would take advantage of the fact that SHRIMP's network interface board already snoops writes on the Xpress bus to implement automatic update. The processor overhead for a monitor command would then be just one cycle in the common case.

To perform a monitoring experiment, user-level software running on the PC initializes the histogram memory (often to all zeroes, but not always) by supplying addresses and data via the monitor's EISA bus interface. The *category* and *threshold* registers are also initialized, as are the bits that indicate whether the monitor should begin in trace or histogram mode, and whether the monitor should toggle into trace mode when a threshold is reached. After this initialization, user software can start the monitor by issuing a special EISA write, and can similarly stop the monitor and read DRAM locations

[1] In extreme situations, threshold overflow interrupts can occur faster than they can be handled. We limit this problem by queuing overflow addresses, and flagging when the queue has overflowed.

using different EISA operations.

4 Software Applications

The mechanisms we have proposed support a wide range of performance monitoring applications, and features like page tags, category registers, and general threshold interrupts facilitate experiments and applications that were difficult or impossible with previous monitoring approaches. This section presents several such applications.

4.1 Using Joint Distributions to Understand Application Behavior

Our first example motivates the use of multi-dimensional histograms to show application behavior on several axes at once. Figure 4 shows histogram output for FFT, a parallel benchmark from the SPLASH-2 suite [27] using 16 processors on 65,536 complex data points. Since a 16-node machine and the performance monitor boards are not completed yet, we gathered this data using a behavioral simulator of the performance monitor, that is integrated into a Tango-Lite simulator [10] of a Shared Virtual Memory (SVM) system being built as part of the SHRIMP project [14]. The performance monitor simulator is a set of C routines that the SVM simulator calls, and that mimic the functionality of the hardware monitor.

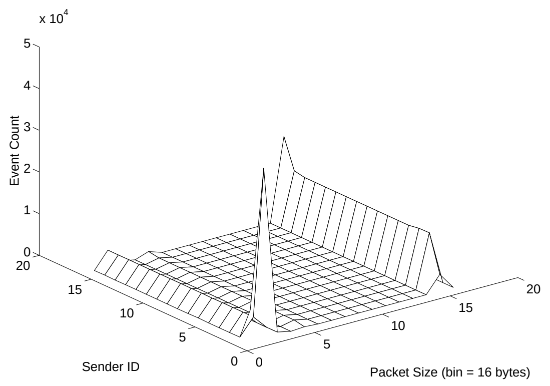

Figure 4: Sender ID and packet size histogram for FFT on SHRIMP's Shared Virtual Memory system.

The plot shows event frequency versus both sender ID and packet size, and represents the summation of results collected at each nodes. Senders are numbered from 0 to 15 and go along the left hand side axis in the horizontal plane. Packet Size is displayed in chunks of 16 bytes, and extends along the right hand side axis in the horizontal plane. The height (z-axis) at each point represents the frequency that a packet of a particular size was sent by a particular processor.

The figure shows the importance of multi-dimensional histograms; the plot allows us to identify two very distinct communication behaviors. First, there is a sharp peak of events for small packet sizes with sender ID equal to 0. This peak corresponds to short synchronization messages that are primarily the responsibility (for this application and SVM implementation) of processor 0. Second, a long, high ridge of events stretches across all sender ID values, for packet sizes of roughly 224 bytes (bin 14). These large packets, nearly uniformly distributed across all processors, correspond to the larger chunks of data being communicated as the FFT is solved. Since the FFT is very well load-balanced, the data communication is nearly even. In our SVM implementation, much of the communication "should" occur in chunks roughly equal to the page size (4096 bytes). In the SHRIMP system however, the EISA bus specification prevents processors from holding the bus for a full 4096 byte transfer; they must back off and re-arbitrate about every 224 bytes. This explains the initially surprising prevalence of packets that size.

This example highlights how multi-dimensional histograms easily show correlations between metrics that are not available with traditional monitors. With a one-dimensional histogram of sender ID counts, we would see a small peak for processor 0, and then roughly even distribution for the rest of the processors. With a one-dimensional histogram of packet sizes, we would see a distribution with peaks at small and large messages, but would not easily be able to attribute the small messages to a particular processor.

4.2 Categorizing Statistics by Code and Data Regions

To understand performance data, one must often be able to attribute measured statistics to particular software causes. Categorizing performance statistics by either code or data regions are orthogonal methods and can each be useful in different cases, depending on whether bottlenecks are more closely tied to code or data structure. The monitor's category register and page tags facilitate this. Some of the monitor's address multiplexer configurations let users choose to have the histogram category and/or page tags form a portion of the address. In this way, the monitor essentially keeps an array of several histograms, each corresponding to different histogram categories and/or page tags.

The histogram category is stored in a register on the performance monitoring board. The category is set by software using monitor commands encoded as memory-mapped I/O writes. Using SHRIMP's automatic update facilities for remote memory-mapped operations, such commands can originate from *any node*. Since the writes occur at user-level, both the runtime overhead and program perturbation incurred by them can be relatively low.

Page tags are stored as part of an existing page-table in each receiving node's network interface [3]. In SHRIMP, all arriving packets require a page-table lookup anyway; for monitoring, the extra page tag bits are sent over from the network interface board to the monitor as each packet arrives. Like the category register, page tag bits are also set by software. For example, one can use the page tags to distinguish heap-allocated memory from static or dynamic data structures, or to distinguish different variables from each other. Updating page tag bits requires *protected*, memory-mapped I/O writes because it updates the SHRIMP page table, which also manages protection for inter-processor communication. The need for protection makes updating the page table tag a more expensive operation than writing the histogram category register. For infrequent tag changes, the overhead is probably not significant. If users change tags frequently (for example, to re-categorize heap memory that is repeatedly allocated and freed), the overhead may be problematic.

One example of a useful, code-oriented statistics categorization might be to place statistics into separate categories according to the procedure the sender or receiver is in when the message is sent. This can help users identify portions of the software that are prone to performance bottlenecks. This functionality is accomplished by allowing either the receiver or a particular sending node to update the monitor's category register via memory-mapped writes.

To evaluate the overhead of per-procedure categorization, we have simulated the execution of the performance monitor within a Tango-Lite simulation of an SVM system running several of the SPLASH-2 benchmarks on SHRIMP. For categorization at this granularity, we issue memory-mapped I/O writes at the beginning and end of each procedure. For programs (like *radix*) with frequent calls to short-duration procedures, command overhead was roughly 62% of the program's unmonitored runtime. For programs with moderate procedure granularity (such as *water-spatial*) overheads were roughly 28%. While far from negligible, this overhead seems tolerable for per-procedure monitoring of many programs. For coarse-grained categorizations (e.g. per-process), the overhead will be considerably lower. Also, in many applications the category may only need to be changed at the beginning of each procedure, process, or phase, rather than both at beginning and end.

4.3 System Debugging Using Threshold Interrupts

Although the monitor has a single threshold register, it can mimic having a separate threshold for each bin if the histogram bins are initialized to the appropriate nonzero values. That is, if we wish every bin i to overflow after T_i events, we put the maximum bin value $2^{40}-1$ in the threshold register, and initialize each bin i to $2^{40}-1-T_i$. (For uses where the actual counts are important, software needs to keep records of these initial values.)

Threshold interrupts, along with other monitor features, provide support for hardware and software debugging. For example, an operating system programmer could assert that in a particular section of the code, only particular processors can write to a memory region. To verify that this is true, the hardware monitor is configured to count events by page tag and sender ID. The pages that should receive no incoming updates from certain processors are given a specific page tag, the interrupt threshold is set to $2^{40}-1$, and the bins that represent the illegal combinations of sender and tag are initialized to $2^{40}-1$. All other bins are initialized to 0.

With this setup, the hardware monitor behaves much like a logic analyzer, waiting to trigger on the occurrence of a particular conjunction of events. If one of the illegal events occurs even once, its initial count causes the threshold to be exceeded, and an interrupt occurs. If a legal event occurs enough times to overflow the counter, that too causes an interrupt. If a trigger condition occurs, the CPU is interrupted, and a software handler decides how (if at all) to respond. This function is especially useful for catching sporadic system bugs, since it allows the code to run at full speed for a long time.

4.4 Adaptive Page Migration Using Hardware Monitor Data

The performance monitor can also give performance feedback to higher-level software *on-the-fly*. For example, in a shared virtual memory (SVM) system built on top of SHRIMP, we can use threshold interrupts and histogram mode to help support adaptive page migration. The SVM system uses SHRIMP's automatic update facility [14], which allows for fine-grained communication between a pair of processors. Higher-level SVM software extends this pairwise mechanism to allow general support for release-consistent shared memory; the system defines a star-shaped communication pattern where all updates for a particular page are sent back to a "home" node; the home node manages the versioning of the page, and sends updates to other nodes as needed.

We are evaluating methods of using performance monitor feedback to migrate the home node of a page adaptively. Performance is improved by choosing the page's home node to be the processor that performs the most updates of the page. Adaptive migration is promising because this fine-grained communication is difficult to predict in advance, may change throughout the execution of the program, and is difficult to quantify without hardware monitoring support.

To implement adaptive migration, we would ideally measure network activity separately for each page, but our design relies on page tags instead. For example, the monitor can keep a three-dimensional histogram of packet counts per page tag, per sender, and per packet size. When a histogram bin's count passes a user-defined threshold, the monitor interrupts the processor. Software then reads in all or part of the histogram and migrates a page if needed. By using different initial values in the histogram memory, we can implement different thresholds for different input combinations. Clearly, the migration overhead must be balanced against its benefits; this trade-off is the subject of ongoing work.

5 Related Work

This section discusses the relationship of SHRIMP's performance monitor to several previously developed projects. For example, the Stanford DASH multiprocessor [17] included a per-cluster histogram-based performance monitor [12]. In the DASH monitor, histogramming is fixed at the time the FPGA was compiled. The histograms allow statistics to be categorized into two user and two operating system categories, or by subsets of the data address bits. There was no provision for runtime flexibility for the data divisions, as allowed by the page tags as we have proposed. A later DASH performance monitor configuration was designed specifically to allow CPU interrupts and OS responses based on observed per-page statistics, but did not allow for general interrupts based on any observed statistic. In contrast, our hardware monitor supports both such specific studies as well as more general monitoring.

Performance monitoring work by Mink *et al.* shows some similarities in approach [19, 21, 22]. Their Multikron and Multikron II hardware monitors are also intended to connect to an I/O bus to monitor activity. Unlike SHRIMP's monitor however, the Multikron monitors are not designed to also connect to a multicomputer network interface, or to collect the sorts of machine-specific network arrival statistics that our monitor does. The earlier TRAMS system connects to the interconnect of an MIMD machine, and counts events filtered by pattern-matching hardware [5, 21]. To our knowledge, however, the TRAMS system's memory design is not intended to provide general multi-dimensional histograms as in the SHRIMP monitor.

The performance monitoring system for Cedar used simple histograms [15], while IBM RP3 used a small set of hardware event counters [4]. The Intel Paragon includes rudimentary per-node counters [23], but cannot measure message latency. Histogram-based hardware monitors were also used to measure uniprocessor performance in the VAX models 11/780 and 8800 [7, 9]. These monitors offered less flexible histogramming, and could not categorize statistics based on data regions or interrupt the processor based on a user-set threshold.

On-chip performance monitors are becoming more common for CPU chips. For example, Intel's Pentium CPU incorporates extensive on-chip monitoring [18]. The Pentium performance counters include information on the number of reads and writes, the number of read misses and write misses, pipeline stalls, TLB misses, etc. Here also, there is no support for categorization of statistics or for selective CPU notification. In contrast, the Alpha 21064 does provide some base level of performance monitoring with selective CPU notification [8]. Its on-chip cache performance counter is used by initializing it to a particular value, and then decrementing it whenever a cache miss occurs; when the counter value reaches zero, the CPU is interrupted.

Some researchers have examined using monitoring information to guide operating system policy decisions. For example, Bershad *et al.* proposed a special-purpose hardware monitor (Cache Miss Lookaside Buffer) that would keep per-page statistics on memory behavior in order to guide operating system decisions about virtual-to-physical page mappings [1]. Chandra *et al.* investigated the potential of dynamically using measured data from a more general purpose hardware performance monitor to guide operating system scheduling and page migration decisions [6]. This approach is closer to ours, but they used an existing performance monitor [12] and focused their attention mainly on determining appropriate operating system policies.

6 Conclusions

This paper has described the design of the hardware and software that make up SHRIMP's performance monitoring system. Our hardware performance monitor design is simple and flexible, and provides novel instrumentation mechanisms such as multi-dimensional packet-based histograms, page tags, histogram categories, and threshold-driven interrupts.

The mechanisms devised for multi-dimensional histogramming and statistics categorization allow operating systems designers and software developers to gather detailed application and workload statistics, and to match statistics with particular software-level constructs. The threshold-based interrupt mechanism is a simple, low-cost and yet powerful mechanism to allow the hardware monitor to interact with operating system and application software. The flexibility and varied functionality of the hardware monitor will help us understand and characterize our current SHRIMP prototype and the measurements and characterizations of SHRIMP will also guide the design of next-generation machines.

Ultimately, a primary goal of any parallel computer is to help programs run fast. In many modern machines like SHRIMP, much of the designers' attention is focused on streamlining the interface between hardware and software, so that user-level processes can take advantage of the large compute power in the underlying machine. The performance monitoring system represents our response to this problem. It allows architects to evaluate the success of their design, while allowing software designers to peer into the black box and understand the detailed behavior of their code in terms of specific performance characteristics.

7 Acknowledgments

This work was supported via a grant from the NSF Experimental Systems Program (MIP-9420653). In addition, Margaret Martonosi is supported in part by an NSF Career Award (CCR-9502516). We would like to thank Kai Li and Ed Felten for several careful readings; their suggested edits improved this paper. Thanks also to the anonymous reviewers for their comments.

References

[1] B. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, Oct. 1994.

[2] M. A. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina. Virtual Memory Mapped Network Interfaces. *IEEE MICRO*, pages 21–28, Feb. 1995.

[3] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21st Annual Int'l. Symp. on Computer Architecture*, pages 142–153, Apr. 1994.

[4] W. Brantley, K. McAuliffe, and T. Ngo. RP3 Performance Monitoring Hardware. In Simmons, Koskela, and Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 35–43. ACM Press, 1989.

[5] R. Carpenter. Performance Measurement Instrumentation at NBS. In M. Simmons, R. Koskela,

and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 159–184. 1989.

[6] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, Oct. 1994.

[7] D. Clark, P. J. Bannon, and J. B. Keller. Measuring VAX 8800 Performance with a Histogram Hardware Monitor. In *Proc. 15th Annual Symp. on Computer Architecture*, pages 176–185, May 1988.

[8] Digital Equipment Corporation. DECChip 21064 RISC Microprocessor Preliminary Data Sheet. Technical report, 1992.

[9] J. Emer and D. Clark. A Characterization of Processor Performance in the VAX-11/780. In *Proc. 11th Annual Int'l. Symp. on Computer Architecture*, pages 301–310, June 1984.

[10] S. R. Goldschmidt. *Simulation of Multiprocessors, Speed and Accuracy*. PhD thesis, Stanford University, June 1993.

[11] S. L. Graham, P. B. Kessler, and M. K. McKusick. An Execution Profiler for Modular Programs. *Software– Practice and Experience*, 13:671–685, Aug. 1983.

[12] M. A. Heinrich. DASH Performance Monitor Hardware Documentation. Stanford University, Unpublished Memo, 1993.

[13] J. L. Hennessy. Distributed Shared Memory: Perspectives on its Development and its Future. In *First ARPA Conference on HPCC*, Aug. 1994.

[14] L. Iftode et al. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *10th International Parallel Processing Symposium*, Apr. 1996.

[15] D. Kuck, E. Davidson, D. Lawrie, et al. The Cedar System and an Initial Performance Study. In *Proc. 20th Int'l Symp. on Computer Architecture*, pages 213–223, May 1993.

[16] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. 21st Int'l Symp. on Computer Architecture*, pages 302–313, Chicago, IL, Apr. 1994.

[17] D. Lenoski, J. Laudon, et al. The DASH Prototype: Logic Overhead and Performance. *IEEE Trans. on Parallel and Distributed Systems*, pages 41–61, Jan. 1993.

[18] T. Mathisen. Pentium secrets. *Byte*, pages 191–192, July 1994.

[19] A. Mink. Operating Principles of Multikron II Performance Instrumentation for MIMD Computers. Technical Report NISTIR 5571, National Institute of Science and Technology, Dec. 1994.

[20] A. Mink and R. Carpenter. A VLSI Chip Set for a Multiprocessor Performance Measurement System. In M. Simmons, R. Koskela, and I. Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization*. 1990.

[21] A. Mink, R. Carpenter, G. Nacht, and J. Roberts. Multiprocessor Performance-Measurement Instrumentation. *IEEE Computer*, pages 63–75, Sept. 1990.

[22] A. Mink and R. J. Carpenter. Operating Principles of Multikron Performance Instrumentation for MIMD Computers. Technical Report NISTIR 4737, National Institute of Science and Technology, Mar. 1992.

[23] J. Rattner. Paragon System. Presentation: *DARPA High Performance Software Conf.*, Jan. 1992.

[24] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. 21st Annual Int'l. Symp. on Computer Architecture*, pages 325–337, Apr. 1994.

[25] J. Roberts, J. Antonishek, and A. Mink. Hybrid Performance Measurement Instrumentation for Loosely-Coupled MIMD Architectures. In *Proc. Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989.

[26] R. Traylor and D. Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proc. Hot Chips '92 Symp.*, Aug. 1992.

[27] S. Woo, M. Ohara, et al. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proc. 22nd Int'l Symp. on Computer Architecture*, pages 24–37, June 1995.