The POWER2 performance monitor

by E. H. Welbon
C. C. Chan-Nui
D. J. Shippy
D. A. Hicks

The POWER2™ performance monitor ("monitor") provides the detailed hardware measurements necessary to study the hardware/software interactions of workloads executed by the POWER2 processor. The monitor is integrated into the processor and is fully software accessible. Of interest is the ability of this monitor to selectively measure specific software processes with minimal perturbation of those processes.

Introduction

Driven by advancing fabrication and miniaturization technologies, advances in RISC processor machine organizations offer new opportunities for enhanced system performance. These opportunities are realized by adapting the fine-grain execution details of software workloads to the special organizational features made possible by advanced technology, as described in [1]. Unfortunately, as systems become more complex and highly integrated, these organizationally dependent opportunities for enhanced performance are often not easily detectable. Consequently, when the performance of a software workload is studied to optimize its performance, performance-critical information concerning the hardware/software interaction is difficult to obtain, and the need for "embedded" performance-monitoring techniques, which do not themselves inhibit performance, is greatly increased.

Important performance measures include instructions executed, elapsed cycles, counts and delays associated with cache and TLB misses, utilization of the various execution elements, and inefficiencies caused by inappropriate instruction scheduling by compilers. Such measures can help to efficiently locate and eliminate performance bottlenecks, and the POWER2™ performance monitor ("monitor") makes them an integral part of POWER2 processor operation.

This paper describes the motivation for the design and implementation of the monitor hardware and the implementation of the monitor access software; it summarizes certain of the measurable events and various monitor usage scenarios. Examples of the application of the measurement data can be found in [2, 3].

We say that a measurement is a hardware measurement if the event being measured both affects performance and is typically invisible to the software. Cache misses are an example of such an event, since cache misses clearly affect system performance and a program cannot generally determine how many cache misses it has encountered.

Hardware measurements can provide direction to the design of future systems by identifying existing bottlenecks, a matter of importance to both system architects and implementors. Just as important, however, is that a good measurement facility can point out ways to improve the performance of existing systems as well. That is, hardware measurement facilities in a given machine can benefit that machine as well as future ones. It is not necessary to justify the expense of hardware measurements on the basis of benefit to future designs only; hardware measurement facilities can also be justified by the improvements afforded to the machine that provides

them. For example, it is useful to be able to identify opportunities for improving instruction scheduling to more efficiently utilize execution elements. Instances of such opportunities are described in [4] and [5]. The development of compiler scheduling algorithms that increase utilization and throughput can be addressed in a more timely manner if it is possible to quickly identify what scheduling opportunities exist.

With an appropriate hardware measurement facility, it is possible to examine the characteristics of large suites of programs more efficiently than otherwise. Since many optimizations performed by compilers are heuristic,[1] it is possible that a given optimization beneficial for a specific program may be inappropriate for other programs. Thus, it is useful to examine the effect of an optimization for a particular program on many other programs to ensure that a specific optimization does not cause a substantial number of other programs to degrade in performance. Improvements to the effectiveness of a system's compiler generally improve overall system performance without necessitating hardware changes.[2]

Of course, hardware measurement is not the only method of understanding system behavior. Simulation can be used to study hardware/software interaction, but simulation is impractical if the systems to be simulated are very large. Also, simulation requires much more time to produce a needed result than does measurement. A final consideration is the difficulty of developing the typical large simulation models required. Indeed, a most effective use of simulation is to study high-level system behavior using measurement data provided at least in part by hardware instrumentation [6].

While simulation is an alternative to hardware measurement for many purposes, a hardware measurement facility that can be accessed efficiently by software offers the opportunity for adaptive system tuning. Given the correct set of measurable events, the system software may be able to sequence system events in such a way as to more fully utilize the system resources. Process scheduling is an area for which adaptation to workload could enhance system performance.

Drawbacks inherent in current RS/6000 measurement facilities

In previous RISC System/6000® (RS/6000) systems, hardware performance information was obtained by special external instrumentation, which was for the most part retrofitted to the processor model under consideration. This was possible in part because the RS/6000 multiple-chip design provided many signals that were needed for functional operation but could also be used to extract performance information [7]. Even so, it was usually the case that derivatives of the functional signals were actually required. To obtain these derivative signals, the functional signals had to be augmented with special-case signals solely for performance monitoring. These signals were provided either by multiplexed functional pins or by pins dedicated to measurement data.

Multiplexing functional pins alternately between functional data and performance data does provide limited access to internal signals that would not ordinarily be externalized. However, such an arrangement typically forces the information bandwidth requirement on such pins to be doubled. This usually means that such pins must be clocked at multiples of the basic CPU clock rate (e.g., double). This becomes increasingly difficult to accomplish as the base CPU clock rate increases. Adding dedicated pins does not have the special bandwidth concern just mentioned, but pins are usually a constrained resource. A consequence is that many important measurements are typically not readily available because of pin limitations. Thus, such special external instrumentation can provide the needed information, but a drawback to such instrumentation is that it requires extra pins or degrades the system CPU clock rate. Moreover, some acquisition device must actually be connected to the provided pins.

System implementors are usually reluctant to design extra components into a board, since these components increase the cost of the system and occupy valuable board space. This usually means that the acquisition device must be externally attached to the system, and some strategy for connecting the required acquisition device must be devised, which entails yet more problems for measurement personnel.

The above considerations typically result in essentially custom (and expensive) instrumentation for each CPU model that is to be instrumented. The result is that such instrumentation is usually designed for only a few system models. Additionally, the custom nature of the instrumentation typically constrains the number of design instances that can actually be physically instrumented.

Desirable monitor characteristics

If the system under test can efficiently access the measurements of its own execution and control the points at which measurements are made, a great simplification of the measurement capability will be afforded to the measurement analysts. A further improvement is a simplified means of adapting the instrumentation to the system.

Assuming that a system can be equipped with instrumentation, there is then the practical matter of defining the points in time and threads of execution that are of interest. Not all execution may be of interest. An analyst studying a numerical analysis application may be

[1] Optimization techniques based on the examination of specific cases may not be generally applicable.

[2] If this process can be carried out in a timely manner, so much the better.

interested in a particular subroutine or even particular loops. An analyst studying a commercial application may want to be able to exclude time spent idle during I/O waits [2].

As indicated, systems are becoming more complex and highly integrated, causing less data to be available as outputs that might allow external hardware to intercept and record the data. In particular, the POWER2 processor is packaged on a multiple-chip module (MCM), and the only available interfaces are for the memory and I/O buses [8, 9].

Regardless of the difficulty increasing packaging densities imposes on hardware measurement, overall system performance continues to become increasingly dependent on a detailed understanding of hardware and software interaction. The availability of a software-accessible hardware measurement facility can be an asset in understanding this interaction and thus in improving the performance of current and future systems.

The instrumentation should be designed so that it can be inexpensively attached, and provide a simple yet effective method to access the information it provides. An example of such capability is the Cray Y-MP [10], which has software-accessible counters. In the Cray monitor, no alteration to the system hardware is required. It was desired that RS/6000 follow-on designs provide similar capabilities.

Implementation of monitor

The POWER2 Central Electronics Complex (CEC) is composed of five basic units (each unit a distinct chip): the fixed-point unit (FXU), the floating-point unit (FPU), the branch unit (or instruction cache unit, ICU), the storage control unit (SCU), and the data cache units (DCUs). Additional details are provided in [8, 11, 12].

As shown in **Figure 1**, the monitor consists of twenty-two 32-bit counters to be used to count performance-related events in CPU and storage. In addition to the counters, a monitor mode control register (MMCR), with a special status bit in the architected machine state register (MSR), allows selective measurement of specific threads of execution. The counters and the MMCR are addressable for read and write via programmed I/O (PIO) operations. Five counters are dedicated to each of the ICU, FPU, SCU, and FXU units. Similarly, each of the four basic units is provided a four-bit control field in the MMCR that selects the set of events to be counted. Thus, for each unit, it is possible to choose any one of sixteen groups of five events for monitoring.

The selection of the events to count for any unit is independent of the selection of those for the other three units. In a special case for the SCU, the setting MMCR(SGA) = 1 will cause the SCU to operate all of the counters for SCU events. This gives much useful information concerning storage operations. In addition to these selectable counters, there are two dedicated counters, a CPU cycle counter and a correctable memory error counter.

The instruction match register (IMR) located in the FXU counts the occurrences of specific instructions. By repetitively cycling through the list of defined instruction codes, it is possible to obtain a sampling of the instruction execution frequencies.

Finally, a bit referred to as the software-programmable event (SPE) allows software to create software events as desired. The number of cycles for which the SPE bit is "1" (set by software) is an event that the FXU counters can count.

Selection of monitoring states

The POWER architecture defines the system-wide machine state register (or MSR, similar to the System/370™ PSW) which is architecturally part of the process state. MSR bit number 29, the process mark (PM) bit, can be used to control the state of the monitor.

Bit number 17 of the MSR, the PR bit, distinguishes between privileged and problem execution states. When PR = 0, all instructions may be executed (this condition is the *privileged state*). When PR = 1, only nonprivileged instructions may be executed (this condition is the *problem state*). Bits MMCR(DP) and MMCR(DU), in conjunction with MSR(PR), can be used to limit monitoring to either privileged state or nonprivileged state as defined by **Figure 2** and **Table 1**.

The MSR(PM) bit can similarly control the state of the monitor. When MSR(PM) = 1, the associated process is said to be *marked*. When MSR(PM) = 0, the associated process is *unmarked*. The state of the monitor with respect to MSR(PM) depends on the setting of MMCR(DMS) and MMCR(DMR) as defined by Figure 2 and Table 1. The setting of the MSR(PM) bit is not altered by the execution of a supervisor call (SVC) instruction, but is set to zero when any other interrupt is fielded, so the monitoring effect of the PM bit is preserved across calls to the kernel. Since the MSR is part of the process state and is saved and restored when processes pause and resume execution, and since only the state of the monitor is affected by the MSR(PM), this bit can be used to selectively qualify processes for monitoring with very low overhead. Thus, the MSR(PM) bit is useful as a nondisruptive method of measuring the performance of specific threads of system execution. The various combinations of the effects are outlined in Table 1.

The settings of DIS, DP, DU, DMS, and DMR bits of the MMCR, coupled with the settings of PM and PR bits of the MSR, provide control of the performance monitor by distinguishing between kernel events and user events.

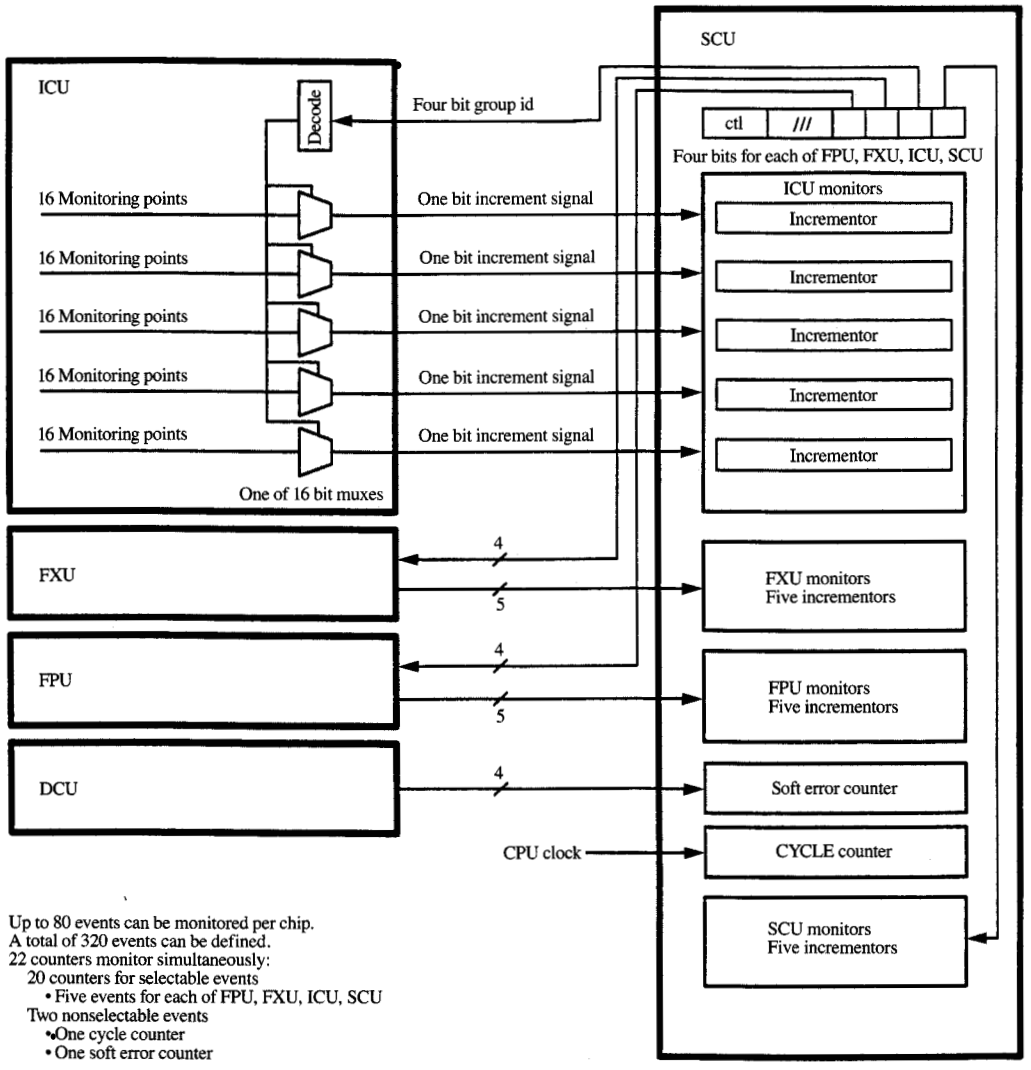


Figure 1

Organization of monitor hardware.

Table 1 shows the set of conditions that can be used to qualify the control of the performance monitor counters.

Thus, with the MMCR the MSR can control the state of the monitor. By incorporating a significant portion of the control of the monitoring function in the MSR, this control becomes part of the process state and is thus saved and restored by the interrupt response activity of the architected hardware. Careful use of this function reduces the intrusiveness of the monitor to negligible levels for most cases of importance.

Selection of observable events

In contrast to the RS/6000 measurement facilities, which provide signals from which external hardware can construct the required performance event in the POWER2

processor, the performance event to be examined can be constructed in the form needed within the unit where it occurs.[3] The list of events to be implemented can therefore be based on what the performance analysts want as opposed to what can be observed given the available signals (even considering the special signals provided to augment the functional signals of the previous RS/6000 designs). As a consequence of this organization, it has been possible for the chip implementors to provide a comparative wealth of event information. A summary of these events is presented in the following four sections.

- *FXU observable events*

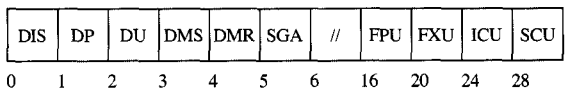
The FXU (fixed-point unit) provides counts of the cycles during which it is inhibited from continuing execution for a variety of reasons. These include branch delays, data cache misses, floating-point interactions, TLB reloads,[4] and the timing of multicycle instructions. Fixed-point instruction counts are provided, including variations in counting the utilization of the internal elements.

A special set of counts identifies cases for which there is opportunity for performance improvement by altering the scheduling of the instruction streams being executed. Related to this is a set of counts that identifies the kind of scheduling opportunities (based on causes of FXU execution delay). Thus, the monitor can help an analyst to identify both the degree and the particulars of the opportunity available from enhanced fixed-point code scheduling.

[3] Generally, the signals necessary to produce the required event are available on the chip. Limited signal availability caused by pin restrictions tends to prohibit this on external instrumentation.

[4] In RS/6000 and POWER2, storage accesses that miss in either the data or the instruction TLB are satisfied by the FXU in hardware and, barring page faults, are transparent to software.

Monitor mode control register



This register is set to 0 on power-up. Reading this register does not change its content.
Bits 6 through 15 are reserved, and always read as 0's. In order to maintain compatibility with future implementations, software must not write nonzero values into bits 6 through 15. The named fields have the following definitions.

DIS: Disable counting unconditionally: This bit, when set to a 1, causes the counters to stop counting unconditionally. The counter values are not changed, only suspended.

DP: Disable counting when in privileged state: This bit, when a 1, causes the counters to stop counting when MSR(PR) is a 0. The counter values are not changed, only suspended.

DU: Disable counting when in problem state: This bit, when a 1, causes the counters to stop counting when MSR(PR) is a 1. The counter values are not changed, only suspended.

DMS: Disable counting when MSR(PM) is set: This bit, when a 1, causes the counters to stop counting when MSR(PM) is a 1. The counter values are not changed, only suspended.

DMR: Disable counting when MSR(PM) is reset: This bit, when a 1, causes the counters to stop counting when MSR(PM) is a 0. The counter values are not changed, only suspended.

SGA: SCU gets all: This bit when a 1, allows the SCU to control all 22 counters; i.e., the SCU gets all of the counters. This bit overrides the FPU, FXU, ICU, and SCU source event set selection.

FPU: MMCR[16:19], four-bit code selecting FPU source event set.

FXU: MMCR[20:23], four-bit code selecting FXU source event set.

ICU: MMCR[24:27], four-bit code selecting ICU source event set.

SCU: MMCR[28:31], four-bit code selecting SCU source event set.

Figure 2

Monitor mode control register format.

Table 1 Defined monitoring states.

DIS	*DP*	*DU*	*DMS*	*DMR*	*Disabled*	*Counted*
0	0	0	0	0	Nothing	Everything
0	0	0	0	1	−PM	Marked (PR = X)
0	0	0	1	0	+PM	Unmarked (PR = X)
0	0	1	0	0	+PR	Privileged (PM = X)
0	1	0	0	0	−PR	Not privileged (PM = X)
0	0	1	0	1	+PR or −PM	Privileged AND marked
0	0	1	1	0	+PR or +PM	Privileged AND unmarked
0	1	0	0	1	−PR or −PM	Not privileged AND marked
0	1	0	1	0	−PR or +PM	Not privileged AND unmarked
X	X	X	1	1	Everything	Nothing
X	1	1	X	X	Everything	Nothing
1	X	X	X	X	Everything	Nothing

- *ICU observable events*

The ICU (branch unit) provides counts of the number of instruction cache and TLB misses that have occurred, the number of instructions fetched, dispatched, and executed, the number of interrupts acknowledged, and a breakdown of the classes of instructions executed (particularly branch instructions). Finally, there are counts that help identify the sources of dispatch and branch delay.

- *FPU observable events*

The FPU (floating-point unit) provides counts that give many details on conditions causing floating-point execution delays, execution element utilizations, and the number of instructions executed and special number processing delays. In addition, there is a breakdown of the classes of instructions executed, and there are counts of cycles causing register-renaming stalls and of various queue-related stall cases.

- *SCU observable events*

The SCU (storage control unit) counts the number of and type of storage operation requests that are presented to it, measures latency for such requests, and measures DMA activity, memory bus activity, and SIO bus activity.

Storage performance is critical in RISC machines [1]; in order to most effectively measure storage-caused delays, a special mode has been defined for the SCU. Ordinarily, the FXU, ICU, FPU, and SCU can operate only five counters each. However, for the SCU there is a special case described in Figure 2 involving the SGA bit of the MMCR that allows the SCU to operate 20 of the 22 counters with strictly storage-control counts. This mode gives a very detailed view of the storage and I/O activity.

Application to system measurement

The POWER2 performance monitor allows monitoring of CPU activity without resorting to sampling. One of the key features of the monitor is the ability to examine particular processes or kernel activity with essentially no overhead. In particular, it is important to note that the monitor can be automatically enabled, either during execution of a particular process or by supervisory activities. Also of note is that supervisory services can be excluded from the measurement process. The same holds true for processes with respect to the MSR(PM) bit. Since it is not difficult to set MSR(PM) = 1 and MMCR(DMR) = 1 for all user processes,[5] it is easy to filter out the user processes and count only operating system tasks, whether kernel or not.[6]

Finally, a single task can also be excluded from measurement. For example, a process whose responsibility it is to manage the monitor could be defined with MSR(PM) = 1; if MMCR(DMS) = 1, in this case all execution is measured except the task that manages the performance monitor. This mode can be used to eliminate the effect of the monitor management task, for example.

Basic monitoring considerations

A 32-bit counter incrementing at a 66-MHz rate will overflow in 65.1 seconds. Thus, as long as the monitor counters are read at least once every 65.1 seconds, no data are lost due to overflow. In particular, if the counters are inspected at every real-time clock interrupt (occurring about once every 10 milliseconds) and every task switch, there is no risk of overflow ambiguity.

If the counters are managed by the operating system at a user's request, it is possible to allow multiplexing of the monitoring function across multiple processes. Since the counters are both read and write, it is possible to give each process the illusion of operating the counter entirely by itself.

To reduce the intrusion incurred by reading the counters, they should be accessed using POWER load/store multiple instructions. In this case,[7] about 60 cycles are required to save 23 general-purpose registers (GPRs), load the MMCR and the 22 counters, store out the MMCR and 22 counters, and then reload 23 GPRs.

If the environment being monitored operates in a steady state, it might prove useful to alternate the events to be counted. On a periodic basis, one could cycle the values of MMCR(FPU), MMCR(FXU), MMCR(ICU), and MMCR(SCU) through the values of 0, 1, 2, $\cdots$, 15. In this way one could sample all events and thereby obtain information from all of the defined event types. Such a procedure was used to obtain the data for [2].

Monitoring all events using a log

If a trace of counts for all process and operating system activity is desired, an algorithm similar to that used by *aixtrace* [13] can be applied. In this algorithm, a log file is constructed for a performance-tracing session. This log file is processed after the tracing session by the postprocessor *rmap*, as described in [13]. This method does not rely on the MSR(PM) for control of the monitor; instead, all processes are monitored. Note that the monitor counters are to be considered process state, just as the general-purpose registers are considered process state. Thus, in order to create the illusion of private counters on a per-thread/process level, it is necessary that the monitor counters be saved and restored as threads and processes stop (or exit) and start (or resume execution). The *aixtrace* trace buffer is used to record the content of the monitor counters. Each entry consists of the process id, the

[5] This is a matter of defining a default value of the MSR to be used as the basis for all processes that are not considered part of the operating system.

[6] Not all operating system execution is carried out by the kernel in privileged mode; some operating system code executes in problem state.

[7] Excluding TLB and page misses, but counting cache misses.

MMCR, the counter values, and the real-time clock at the time that the counters are read. As this buffer fills, it is written to a mass storage device if the accumulated data are to be postprocessed by *rmap*.

The save/restore algorithm for the monitor is as follows:

1. At the point in time that a monitored process is stopped, record the current monitoring mode, counts, and process ID into a system-wide buffer in a storage area designated for this purpose.
2. Immediately before starting the next process, record the mode and counts that the operating system has accumulated. Set the monitoring mode and events of interest for the process to be started or resumed, and clear the residual counts in the counters.
3. Start or resume the process (usually with an RFI or RFSVC instruction).

The implementation of this procedure is discussed in a subsequent section of this paper.

Monitoring one process

Almost all performance studies focus on a single process, usually a benchmark such as li from the SPEC™ benchmark suite [14]. In such a case, the MMCR and the MSR(PM) bits can be used to great advantage in collecting performance data.

The benchmark in question has both MSR(PM) and MMCR(DMR) set to 1. All other processes executing have their respective MSR(PM) bits set to 0. As a consequence, only the benchmark process with MSR(PM) = 1 is able to alter the performance counters. Thus, the performance counters do not have to be saved or restored, eliminating the attendant overhead.

As mentioned earlier, the counters may "roll over" as often as once every 65 seconds. Thus, the operating system must schedule an event that will save the performance counters at a frequency consistent with this. Also, the monitor control software must supply certain managerial functions for the monitor, because reading and writing the MMCR and the MSR are privileged operations. In general, the monitor is a serially reusable (i.e., the monitor contains state information) resource and must be managed to some extent to ensure that it is used in a reliable manner.

Monitoring a set of processes

It is possible to monitor the cumulative effects of any set of the executing processes using the above technique. To do this, the control software must be able to recognize what subset is to be monitored. When the subset has been determined, each process of the subset will have its MSR(PM) set to 1.

As long as a consistent set of parameters to be measured is specified across the process set, each time a marked process (from the set of processes to be monitored) commences execution, the MSR for that process is reloaded with the previous state, which enables the monitor. [In AIX there is a special case called a "process group," consisting of a parent process (the group leader) and all of its progeny (processes that it creates via fork, etc.). In this case, each child inherits its MSR from its immediate parent. Hence, if the group leader (i.e., the root process of the group) is marked, all of the children will likewise be marked and hence easily monitored. No special effort to achieve this behavior is required, since it capitalizes on an aspect of AIX architecture.] The data stream emanating from the counters can be treated in the same way as the data stream from a single process.

Monitoring segments of code with well-defined entries and exits

If a segment of code has exactly one entry point, it is possible to monitor the events that occur within the segment by placing a call to enable monitoring at the entry point and placing a trap instruction at each possible exit point. While the call to enable monitoring and the trap to disable monitoring incur some overhead, this overhead does not affect the measurements made on the code segment in question.[8] The implementation of such capability is discussed in a subsequent section of this paper.

System-level monitor software

The software for the POWER2 performance monitor has evolved to support two types of tasks, providing data for visualization software and producing counter data with as little intrusion as possible. This section briefly describes both software methods and their future potential.

Data for visualization software such as *pv* [15] is provided through the *aixtrace* facility [13]. This allows the visualization software to correlate the data provided by the performance monitor with the normal trace data in order to provide more informative displays. The counter data are inserted into the trace as an *aixtrace* record every *n* software clock ticks, or when the process dispatched is different from the current process. This allows report generators that postprocess the trace to produce per-process performance statistics.

Calls which request the capture of information are embedded at particular and significant locations in the software. These calls are referred to as "hooks." Each distinct hook has a unique identifier, the hook ID, which is used to distinguish between the various hooks. Hooks

[8] Except for possibly causing a small number of extra cache misses and some initial disruption of the pipelines.

may be system- or user-defined and may be disabled and enabled as desired. Each is designed to pass hook-specific data. A purpose of *aixtrace* is to record the data passed by these hook invocations.

POWER2 performance monitor *aixtrace* support is implemented using a kernel extension which attaches itself to the operating system call that puts data into the *aixtrace* trace buffer. The monitor counters can be dumped when either system- or user-specified trace hooks are triggered. This gives the report generators the ability to fine-tune their analysis, with the finest granularity being that of the trace hooks that are enabled. The extension analyzes the arguments of the hook and, on the basis of the hook ID, some state variables, and the link register, determines whether it is an appropriate time to dump the counters. If so, the kernel extension calls the trace routine on its own behalf, inserting the counter data into the trace.

This implementation is a fairly flexible one, since the kernel extension provides the ability to specify which hooks cause the monitor counters to be recorded. Also, additional software logic may be added without having to rebuild the kernel. The major drawback of this method is that the *aixtrace* facility and the appropriate kernel extension must be active. This means that depending on the number of hooks enabled, data gathering is slightly intrusive (in the neighborhood of 10%).

A more efficient way of implementing this functionality would be to modify the operating system kernel directly. This would significantly reduce the overhead incurred by the kernel extension. A second method addresses this problem in a different way.

Monitor command line tools

The second software method, the POWER2 performance monitor command line tools (CLT), was created to provide a simple way to acquire the performance monitor counters. It consists of a set of command line programs and system calls that allow the user to manipulate the performance monitor control registers (the MMCR and IMR) and to periodically write the values of the counters to the operating system *standard out* stream or a *file.*

If a program runs in a short time (less than 65 seconds, given a 66-MHz processor clock), it is possible to reset the counters, run the program, and acquire the counters. One can then monitor the program with no intrusion at all. For longer programs, it is necessary to save the counters periodically to prevent them from overflowing. One of the tools operates somewhat like *iostat* [13] and writes the counters at a frequency specified by the user. When this program or an analogous one is run, the amount of intrusion is roughly equivalent to a context switch each time the counters are recorded. To gain finer granularity in the measuring of a program, it is possible to instrument the program with function calls to manipulate the control registers. Thus, one can analyze only certain portions of the code and gather performance data just for those sections.

Access to the performance monitor hardware is provided through a kernel extension. The extension provides a set of system calls which manipulate the control register and allow the resetting or retrieval of the counters. The command line tools simply call functions available in the kernel extension. The major drawback to using this technique for generating data is that the current operating system kernel does not preserve the MSR(PM) bit, and per-process tracing is therefore not available. It is, however, possible to modify the operating system kernel to preserve this bit; if this is done, per-process tracing is possible. Effort is underway to accomplish this.

Monitor data reduction tools

The last component of software for the monitor involves the data reduction. Several tools are available to reduce the data generated by the MMCR to useful information. The visualization tool *pv* uses a filter that converts the *aixtrace* data, including the additional monitor data, into a *pv* data stream, allowing *pv* to display informative graphs of the state and performance of the machine. Because the trace stream contains large amounts of operating system data in addition to the counters, *pv* is able to give a much richer display than would otherwise be possible. The coupling of the monitor data to the operating system data has proven most useful [2].

To easily generate interesting values from the data provided by means of the monitor command line tools, a set of postprocessors was developed. Because it is possible to generate interesting values in a variety of ways, it is nearly impossible to predict what derived values a particular user might find most interesting. Therefore, the postprocessor uses a user-specified input-rule set to determine how to arrive at a given set of results. The postprocessor reduces the input to these values, complete with statistics. A secondary postprocessor uses these results with a different set of rules to produce more complex relationships. This is useful when programs have large steady-state periods. With such programs, it is possible during these periods to change the MMCR value in order to get a broader set of counters than normally available. With the expanded set of counters, it is possible to derive values not otherwise obtainable.

This set of programs has been successfully used to analyze several benchmarks, including TPC [2, 3].

Summary

The POWER2 performance monitor provides a set of hardware measures that are important to software developers tuning operating system and application software as well as to system designers responsible for the

development of new systems. The events monitored include important measures involving storage system performance and compiler instruction scheduling, both critical to good system performance.

Acknowledgments

We would like to thank several members of the AIX® and System Performance groups, and the Engineering Center for their help in developing the monitor. In addition to authors Shippy and Hicks, Larry Thatcher of the FXU design team, Robert Golla and Jama Barreh of the ICU design team, and Richard Fry of the FPU design team provided pragmatic suggestions useful in defining events to be measured. Peter Markstein, on behalf of the Compiler design team, provided insight into useful floating-point measures. Architects Ed Silha and John O'Quin assisted in the definition of the measurement control. Author Chan-Nui developed the support software, aided by useful numerical analysis insights provided by Processor Architects Steve White and Sohel Saiyed. Michael Fortin, Randy Heisch, and John Iacoletti provided consultation for the kernel extension definition and implementation. Maurice Franklin worked out practical experimental designs that provided measurement data initially considered unobtainable. Bryan Rosenburg and Doug Kimelman provided support in *pv*.

References

1. John Cocke and V. Markstein, "The Evolution of RISC Technology at IBM," *IBM J. Res. Develop.* **34,** No. 1, 4–11 (January 1990).
2. M. T. Franklin and E. H. Welbon, "POWER2: Performance Measurement and Analysis of TPC-C," *Digest of Papers, COMPCON Spring '94*, Order No. 5380-02, IEEE Computer Society Press, Los Alamitos, CA, pp. 399–404.
3. M. T. Franklin, W. P. Alexander, R. Jauhari, A. M. G. Maynard, and B. R. Olszewski, "Commercial Workload Performance in the IBM POWER2 RISC System/6000 Processor," *IBM J. Res. Develop.* **38,** No. 5, 555–561 (September 1994, this issue).
4. H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 85–92 (January 1990).
5. M. C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM J. Res. Develop.* **34,** No. 1, 93–97 (January 1990).
6. Syed Z. Pasha and E. H. Welbon, "Performance-Directed Design Guidance Using Simulation," *IBM RISC System/6000 Technology*, Order No. SA23-2619, available through IBM branch offices.
7. G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 37–58 (January 1990).
8. S. W. White and S. Dhawan, "The Next Generation of the RISC System/6000 Family: POWER2," *IBM J. Res. Develop.* **38,** No. 5, 493–502 (September 1994, this issue).
9. R. S. Reddy and D. Galvin, "RIOS2 Packaging Technology," *IBM High Performance RISC System/6000 Technology*, Order No. SA23-2737-00; available through IBM branch offices.
10. *Cray X-MP Computer Systems 4—Processor Mainframe Reference Manual*, Order No. HR-0097, Cray Research, Inc., Minneapolis, MN.
11. D. J. Shippy and T. W. Griffith, Jr., "The IBM POWER2 Fixed-Point, Data Cache, and Storage Control Units," *IBM J. Res. Develop.* **38,** No. 5, 503–524 (September 1994, this issue).
12. T. N. Hicks, R. E. Fry, and P. E. Harvey, "IBM POWER2 Floating-Point Unit Architecture and Implementation," *IBM J. Res. Develop.* **38,** No. 5, 525–536 (September 1994, this issue).
13. *IBM AIX Version 3.2 for RISC System/6000, Performance Monitoring and Tuning Guide*, Order No. SC23-2365, available through IBM branch offices.
14. K. M. Dixit, "Overview of the SPEC Benchmarks," *The Benchmark Handbook for Database and Transaction Processing Systems*, Jim Gray, Ed., Morgan Kaufman Publishers, Inc., San Mateo, CA, 1994, pp. 489–524.
15. D. N. Kimelman and T. A. Ngo, "The RP3 Program Visualization Environment," *IBM J. Res. Develop.* **35,** No. 5/6, 635–651 (September/November 1991).

Received August 3, 1993; accepted for publication March 28, 1994

Edward H. Welbon *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (welbon@austin.ibm.com).* Mr. Welbon received a B.S. degree in electrical engineering in 1978 and an M.S. degree in electrical engineering in 1979, both from the University of Miami at Coral Gables, FL. In 1979 Mr. Welbon joined IBM Boca Raton, where he worked on a S/1 single-chip minicomputer implementation. In 1986 he transferred to IBM Austin to work on the RISC System/6000 project. Mr. Welbon is an Advisory Engineer, has numerous publications, and has received an IBM Outstanding Technical Achievement Award.

Christopher C. Chan-Nui *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (channui@austin.ibm.com).* Mr. Chan-Nui received a B.S. degree in computer engineering from the University of Washington in 1992. That same year he joined the Architecture and Performance group at IBM Austin and is now an Associate Programmer. Mr. Chan-Nui has spent the last year developing software tools for hardware performance monitoring and data analysis.

David J. Shippy *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (shippy@austin.ibm.com).* Mr. Shippy is currently working on the design and architecture of future PowerPC microprocessors. Prior to this he worked on the design of the POWER2 microprocessor. He started his career with IBM working on midrange System/370 processor designs. Prior to joining IBM, Mr. Shippy worked on the

hardware and software design of multiprocessor computer systems with Emerson Electric, Electronics and Space Division. He holds a B.S. degree in electrical engineering from the University of Kentucky and an M.S. degree in computer engineering from Syracuse University. Mr. Shippy has numerous patents and technical publications.

Dwain A. Hicks *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (dwain@austin.ibm.com).*
Mr. Hicks received a B.S. degree in electrical engineering from New Mexico State University in 1984 and an M.S. degree in electrical engineering from Columbia University in 1987. In 1984 he joined AT&T Bell Laboratory, where he worked in the Microsystems group. Mr. Hicks joined the IBM Corporation in 1987; he is an Advisory Engineer in the Advanced Processor Design group in Austin. He was a team leader in RS/6000 data cache design and continues to work on projects in the area of cache design.