A programmer's view of performance monitoring in the PowerPC microprocessor

by F. E. Levine
C. P. Roth

Performance monitor (PM) support in on-chip PowerPC® microprocessors is used to analyze processor, software, and system attributes for a variety of workloads. The interface to the PowerPC 604® microprocessor, which we abbreviate "604," has been externalized to end users. We discuss the enhanced PM support available in an upgrade of the 604, the PowerPC 604e™ microprocessor, which we abbreviate "604e." We discuss the challenges related to the externalization of the PM support as it relates to other PowerPC processors not derived from the 604 and briefly contrast these PMs with other PMs. We also describe an application programming interface (API) to the on-chip PM support, its design methodology, and its usage considerations, intended to meet these challenges.

Introduction

Performance monitors (PMs), which provide detailed processor and system data, have traditionally been viewed as proprietary hardware luxuries that are available only to large multichip processors. With the rapid evolution of microprocessor technology, its complexity has matched or exceeded that of the old mainframe technology. A full understanding of system characteristics for complex workloads is unobtainable without some additional hardware assistance. The use of test instruments attached to the external processor interface has not been entirely satisfactory. Such instruments cannot determine the nature of the internal operations of a processor, and they cannot distinguish among instructions executing in the processor. Because this approach requires extra hardware and significant expertise, even a simple bus trace may not be practical in most development environments. Test instruments designed to probe the internal components of a processor are typically considered prohibitively expensive because of the difficulty associated with monitoring the many buses and probe points of complex processor systems that employ pipelines, instruction prefetching, data buffering, and more than one level of memory hierarchy within the processors. A common approach for providing performance data is to change or instrument the software. This approach, however, significantly affects the path of execution and may invalidate any results collected. For these reasons, the concerns related to performance

0018-8646/97/$5.00 © 1997 IBM

analysis have been recognized by other processor manufacturers such as Intel**, which included a PM in the Pentium** and Pentium Pro**.

The externalization of the PM interface for the high-volume PowerPC 604* [1] microprocessor has changed the perception that such interfaces are proprietary, and has raised the standard of excellence in this area. It has also encouraged other manufacturers to disclose their PM interfaces, which had not previously been described in their user manuals. The functionality of the 604 PM has generally been regarded as excellent, and its widespread use has ensured that most new PowerPC* processors intended for use in servers, workstations, or personal computers will have an on-chip PM and the documentation required to access the PM. We describe some enhancements that have been incorporated in an update to the 604, the 604e. Although many capabilities and the means of accessing them are similar among different PowerPC processor families, there are some areas of difference. We discuss the PM application programming interface (API) approach to address the areas of difference in performance monitoring support that are found between different PowerPC processors.

Evolution of performance monitors in POWER and PowerPC processors

Early versions of RISC System/6000* POWER processors had no monitoring support in the processor. In 1991, cards were created to attach and monitor processor bus activity.

In 1992, the RISC System/6000 POWER2* [2] processor, which we refer to as the "P2," had 22 counters integrated in four units, which were implemented as separate chips. The P2 PM provided up to 16 groups for each unit. A selected group from a unit would allow for the counting of five events (fixed for each group) on five counters. The selection of a group for a unit allowed for the counting of five extremely low-level (at most one count per cycle) events that occurred within the unit. It took many counters to determine the total number of instructions dispatched.

The design of the P2 provides counting control based on a process or thread context and a user-versus-system execution. For the process or thread context, the hardware interface allows a bit in the machine status register (MSR) called the PMM bit to control counting. The operating system must support the setting and propagation of the bit as part of the process or thread context in order to use this support. The hardware interface provides support to prohibit counting while the processor is in user mode or in system mode. The supervisor-versus-application state is also maintained in the MSR.

In 1993, revision 2.0 of the 604 was sent to be manufactured with an on-chip PM built into the design. The 604 PM contained the same support as the P2 had for the PMM bit and the ability to suppress counting when operating in application or system mode.

In 1995, the 604e was released to manufacturing with a more extensive PM than that provided for the 604. The enhancement includes two more counters and about twice as many events. Compared to the P2, the 604 and 604e contain a rich set of functional enhancements, many of which are described in this paper.

In 1997, the RISC System/6000 POWER2SC*, which we call the "P2SC," was shipped in systems; thirty-two were incorporated in "Deep Blue," the computer that defeated G. Kasparov in a chess match sponsored by IBM. The P2SC contains a performance monitor derived from that of the P2. The P2SC has two dedicated counters, of which one always counts cycles (as was supported in the P2) and the other counts instructions completed. The P2SC also has five counters which can select events from each of the five event positions in any group in any unit. Unlike the P2, the P2SC supports events that increment by more than one per cycle. The P2 and P2SC allow counting but do not support some of the additional features incorporated into the PowerPC processors (e.g., taking an interrupt when a counter is negative, that is, when bit zero is on). With the P2 and P2SC, care must be taken to read the counters before they wrap to avoid incorrectly accumulating counts.

In 1997, the PowerPC 620* microprocessor is under development, with a PM that has features similar to the 604e and some additional features and counters.

Use of the PM

Reference [2] provides a description of the P2 PM and its intended use, while Reference [1] describes the 604 PM and details its intended use. Reference [3] also describes the 604 PM, including specific examples related to event sampling, thresholding, and symmetric multiprocessor (SMP) analysis. Reference [4] shows how the 604 PM can be used to examine and contrast the effects of hardware variations on system performance, while Reference [5] describes the intended use of the 604e PM and provides some additional insight into items mentioned in this paper.

The sampling techniques described in Reference [6] are instrumental to most PM applications. Sampling can be used to identify specific areas of poor performance, which may be minimized in either the hardware or the software. For example, by providing for a determination of the amount and types of misaligned accesses in relationship to aligned accesses, one can determine the penalty associated with leaving the data misaligned. Use of the profiling capability and the information on misaligned accesses allows one to estimate the performance improvement expected from changing the code. We can also use these data to determine whether it is worth improving the processor for a future version.

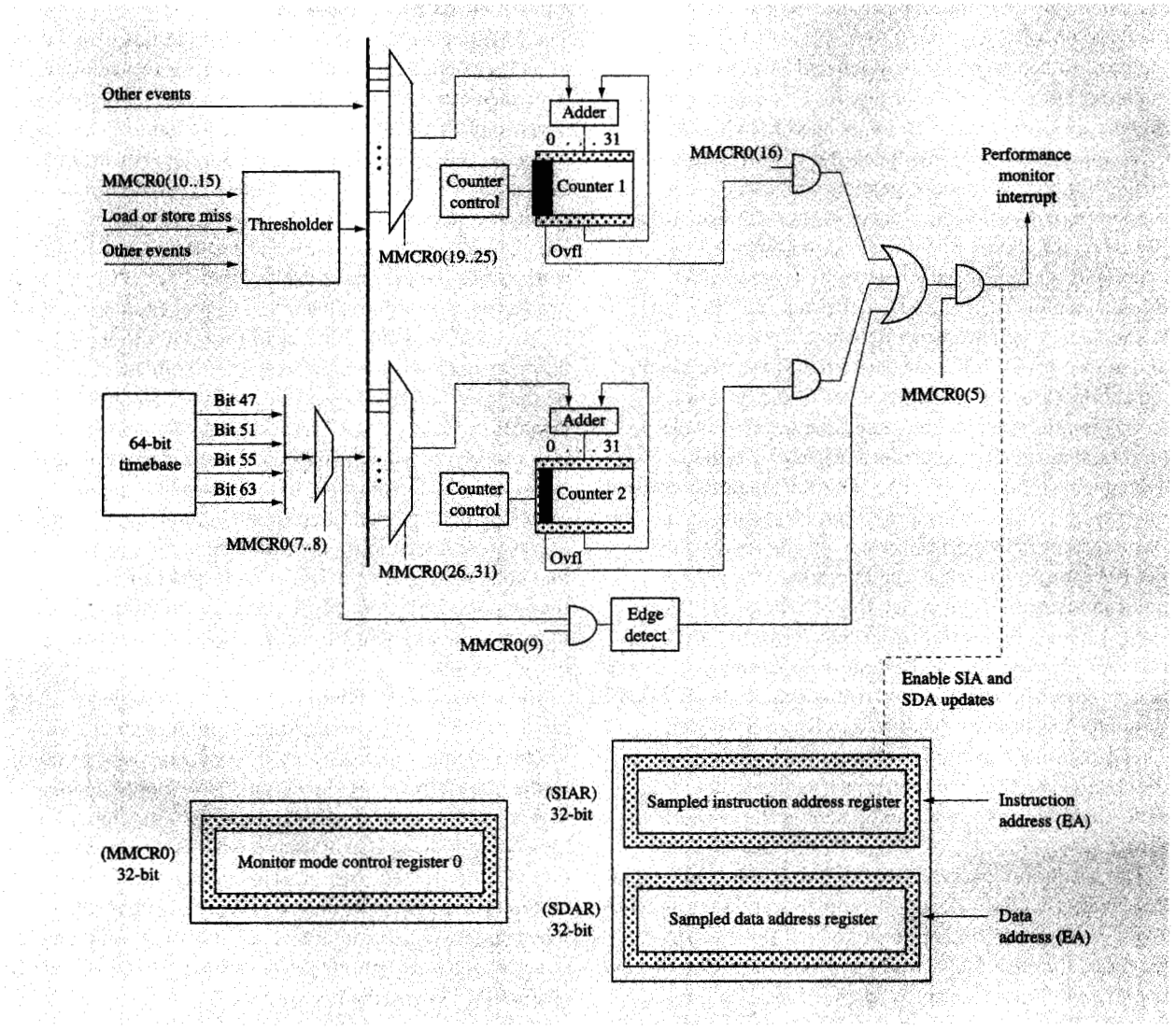

Figure 1

Design of the 604 PM.

A large portion of any code running on a processor is likely to have been written for a different machine and even a different machine architecture. Different machines have different performance characteristics related to the alignment of the data being accessed. However, when some of the systems were being developed with a version of the 604, a simple approach of repeatedly running a specific application with different events showed that the time involved in handling access to misaligned data was causing too much overhead. Because this problem was occurring on "legacy" code and code running under some type of emulation, the decision was made to improve performance related to the handling of unaligned data in an update of the 604. The data were collected by simply accumulating all values of counters providing this information without the need for sampling. The application ran quickly enough to avoid the problem of having the counters overflow.

Features of the 604 PM

The 604 on-chip performance monitoring support provides counting control based on a process or thread context, user-versus-system execution, interrupt signaling, and a counter dependency.

The 604 PM (**Figure 1**) provides support for a PM interrupt. The monitor can be programmed to signal an

exception when a counter is negative or if a selected timebase bit switches from 0 to 1. The timebase is intended to be synchronized among all processors in a multiprocessor system and is discussed in *The PowerPC Architecture** [7]. The exception is masked by the MSR exception-enable (EE) bit. When the EE bit is on and no higher-priority exception is pending, the exception is supported by transferring control to the software at the PM interrupt vector, which is at the real memory address 0xf00. When a PM interrupt is signaled, the sampled instruction address register (SIAR) is set to the address of an instruction which is executing. Simultaneously with the setting of the SIAR, the sampled data address register (SDAR) is set to the operand of an instruction which is executing. The priority of the performance monitor interrupt is higher than the decrementer interrupt priority and lower than the external interrupt priority. This priority allows software to take the PM interrupt before a task switch occurs. At the time the PM interrupt is taken, the operating system has interrupted the task for which the SIAR and SDAR are valid, allowing the operating system to identify the correct task. Another use for the PM interrupt is simply to ensure that all internally maintained counts include the fact that a counter has become negative and is about to overflow. This is typically handled by adding the value of the counters at the time the interrupt is taken to the software-maintained internal values and resetting the counters to zero.

The hardware interface supports the function of having PM counter n (PMCn), $n > 1$, wait to start counting until PMC1 is negative (bit zero on). We call this the *trigger* function, although the 604 user's manual identifies this as the *discount* function. This is intended to allow counting to start after a specific condition has occurred a selectable number of times.

Threshold support is provided in the 604 PM for loads and stores. The threshold value is identified by a field in monitor mode control register 0 (MMCR0). The threshold can be used to analyze the true cost of queueing [4]. The 604 allows the threshold to be gated by an external pin, which was intended to support lateral intervention (data in another processor's cache in an SMP system).

The ability to control the signaling of exceptions based on a counter value and the setting of the SIAR and the SDAR provide a profiling capability whereby the technique of sampling can be used to identify the frequency of occurrence of the monitored events as it relates to specific pieces of code. For any item that can be counted, this profiling capability provides a means to identify the pieces of code in which the measured item occurs and its relative frequency of occurrence. This information provides a valuable tool for operating system developers and processor developers.

Features of the 604e PM

The PM support in the first release of the 604 is described in its user manual [1], which documents a two-counter implementation with 40 unique events. The primary purpose of the 604 PM is to provide an aid to operating system developers and application developers who wish to tune software. The 604e PM (**Figure 2**) provides more counters and more events. Many of the new events are intended to provide more insight into the internal workings of the processor itself. Other PowerPC processors, with different internal characteristics intended for different system platforms, are expected to have different performance-monitoring requirements. For this reason, one should not expect consistency among different PowerPC processors with respect to the number of counters or the actual events available for monitoring.

PM support for the 604e has a total of 111 unique events that can be measured. The additional counters and events were added in an upward-compatible manner, so that code written for the two-counter 604 can run on the four-counter 604e. We use the phrase "additional support" to indicate events that can be measured in the 604e but not in the 604.

The 604e provides additional support to study a program's memory access patterns and interaction with a system's memory hierarchy. Additional support was added to the already rich support in the 604 because memory access patterns can be adjusted by system developers, and they typically have a significant impact on the performance of a given workload. Also, understanding memory hierarchy behavior aids in developing algorithms that schedule and/or partition tasks, as well as distribute and structure data for optimizing the system. This is especially true in SMP environments, where data may be needed by one processor but held by another. For these reasons, events related to the MESI [modified, exclusive, shared, invalid (cache consistency protocol)] protocol are available for analysis.

The length of time it takes to service a queue has been added via specific implementation of Little's law: "The average time a customer spends in a system equals the average number of customers in the system divided by the average rate at which a customer enters or leaves the system." Applying this law to the processor system, the average time required to process an instruction (e.g., a load or store) equals the average number of instructions held in the instruction queue divided by the average number of instructions in the queue. By knowing the average processing time for instructions, designers develop a better understanding of the actual time spent executing instructions. Using this information, system designers may decide to change the hardware or software elements to increase system performance in the most appropriate manner.

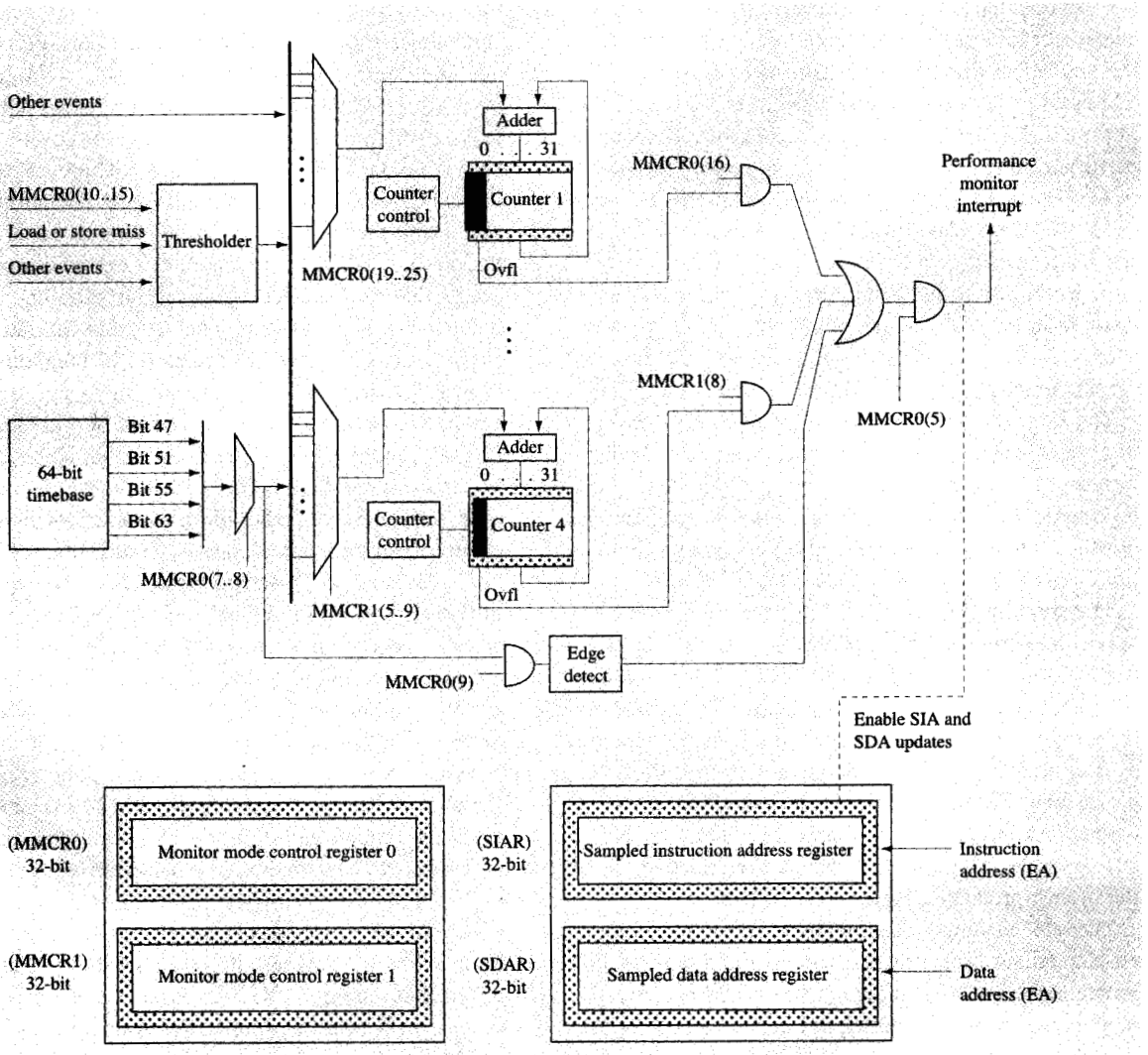

Figure 2

Design of the 604e PM.

Because PowerPC processors are able to execute instructions out of order and speculatively, it is possible that forward progress may be made while a particular unit is waiting for data. If a unit has some work to perform but is unable to do any useful work, we say that that unit is "stalled." If a unit has no work to perform because it has no instructions to execute, we say that that unit is "idle." Providing information related to idles, stalls, and the general efficiency of individual units not only helps in the understanding of the processor internals, but also provides information that may be used for software tuning (including compiler-scheduling algorithms) and system design considerations. For example, information related to the dispatch unit may be used to determine whether there is too much hardware support for dispatching a specified number of instructions in a single cycle. By reducing the hardware support and dispatching fewer instructions in a single cycle, it may be possible to increase the cycle frequency and speed up the overall performance. Such action may provide for feedback into compiler-scheduling algorithms as well as feedback into future processor design. Stalls identify processor resource deficiencies; for example, one can determine whether the processor provides enough reorder buffers.

In the 604e, support was added to count the number of occurrences of certain classes of instructions (e.g.,

instructions that serialize execution). By identifying the amount of time spent executing a specified instruction, the number of occurrences of the instruction, and the addresses of the occurrences, one can use this collected information to determine the expected improvement in performance for proposed changes regarding the handling of the instruction in either the software or the hardware.

The 604e provides the ability to count matches at a specific instruction address. By selecting this event to measure in PMC1 and using the *trigger* function, called *discount* in the 604 manual, one has the ability to start counting when a specified instruction has been executed a specific number of times.

The 604e also provides the ability to count the number of cycles spent while interrupts are inhibited or while an interrupt is pending and interrupts are inhibited. This information can be used to pinpoint software anomalies, in which interrupts are inhibited for too long a period of time.

The speculative execution of instructions provides for an opportunity to do useful work by prefetching results required to avoid bottlenecks. If the branch-prediction logic is effective, many stalls can be avoided or minimized. If the branch-prediction logic is not effective, the extra work of fetching unneeded data could even cause the processor to be less effective than it would be if speculative execution were simply avoided altogether. The 604e provides support to determine the effectiveness of the branch-prediction logic.

The 604e also provides the ability to count and identify misaligned accesses, which facilitates the discovery of the problem previously described. The 604 did not directly support the counting of misaligned data accesses, but it did provide other information which pointed to the problem. For example, it double-counts misaligned loads, and the time spent executing loads is available with the threshold function.

Processor differences

As we have shown, the 604e can measure a wide variety of events to understand the internals of the processor and to relate these internals to specific pieces of code. Other processors that are compliant with the PowerPC architecture have also developed performance monitor facilities. Because these processors have different internal processing algorithms, different events may be selected for processing. Since the PM facility was not included in the PowerPC architecture at the time of development of the 604 and the 620, a significant interface difference occurred in the PM facility between the 32-bit word processors and the 64-bit word processors. Specifically, the special-purpose register (SPR) numbers for the PM registers used in the 32-bit machine are different from those used in the 64-bit machines. The P2 PM and the P2SC PM use input/output (I/O) space, that is, specific addresses to communicate with the software, instead of SPRs. An updated internal version of the PowerPC architecture now includes the PM facility in an appendix as an optional feature which does not require processor developers to follow the described implementation. However, the PowerPC architecture does identify the PM interrupt vector and the currently used PM SPRs as being used by the optional PM facility. The number of counters, the events that can be selected on specific counters, the number of events, and the types of events are all expected to vary. Although there is a basic set of functions, one can expect the actual specific functions supported to vary among the different processor classes. In fact, the PowerPC architecture appendix describes some features which are currently incorporated in the 620, but not in the 604e. This appendix describes a mode in which all of the counters can maintain a cycle-ordered history of occurrences of the selected events. On a given cycle, the PMC*n* register is shifted left and the low-order bit is set on, if and only if one or more of the selected events assigned to each PMC*n* has occurred. With multiple counters running in this mode, one can observe a cycle-ordered relationship among the selected events. It also describes the instruction address break-point register (IABR), which can be used by the PM to prohibit counting until execution occurs at a specific address in a process where counting would otherwise have been enabled.

The PM API approach to conceal differences

To alleviate the software problems related to the differences among the various PowerPC processors, the development of a PM application programming interface (API) is underway.

The PM API is designed to conceal PM processor differences from applications; to increase the flexibility of processor implementations; to facilitate the development of tools to use the PM functions; to promote the use of the PM; and to create an open interface (available to all operating systems on PowerPC systems).

Requirements and objectives of the PM API

Figure 3 depicts at a high level the placement of the PM API in the software hierarchy. The PM API software is being developed to work as an internal IBM tool which runs under the AIX* Operating System Version 4.2. Although there is no commitment from IBM to make the PM API a product or to provide any support for this tool, the source and/or object for this tool is currently made available (at no charge without any warranties or support) to other interested parties. A version is currently available and is under test internally. This preliminary version is available externally for individuals willing to agree to the no-charge, no-warranty, and no-support license agreement.

• *Requirements*

The primary requirement of the PM API is to provide an access mechanism to the on-chip performance-monitoring functions. The lowest layer of support should be provided as a loadable kernel extension for those operating systems that provide that type of support, or as part of the shipped operating system for those systems that do not provide loadable privileged state code.

The support must provide a means to treat the performance monitor counters as a serial reusable resource so that the concurrent running of two applications will not change the integrity of the counting process.

• *Objectives*

The primary objective of creating an API to the on-chip performance-monitoring facility is to conceal the processor differences, encouraging tool writers to access the on-chip performance-monitoring facilities. If this is done properly, tools written to the PM API should work "as is" on other processors with an updated API support. The design should plan for support to handle items that may change between different processors, such as SPR numbers (or other means to communicate with the on-chip performance-monitoring facility), the number of counters, the number of events, event-selection values, and functions supported.

Another objective is for the PM API to be portable and available externally for adaptation to other operating systems. By making the source and thus the interface available to other operating systems, we have an "open interface," which should promote the development of shipped and supported tools that use the interface. The availability of supported tools should encourage additional functionality on new processors and increase the value of the performance monitor support.

Design and definition considerations for the PM API

• *Design methodology*

We describe the high-level design methodology intended to meet the requirements and objectives for the PM API.

The actual update of the hardware, defined as SPRs on PowerPC processors, must be done in the supervisor or privileged state and is supported by routines in a loadable kernel extension. In operating systems where loadable kernel extensions are not supported, these routines must be provided as part of the operating system support. This "privileged" code provides interfaces to commonly required functions. The layer for this support is called the PM kernel extension (PMkex). In AIX, all data accessed by the interrupt handler can be accessed only in the supervisor state. Application requests to the PM API go through an application or library layer that validates the request, ensures the availability of the resource, and passes the request to the kernel extension layer. Application data are imported into the PMkex layer using the AIX operating system routine, *copyin*. Data are exported from the PMkex layer to the application layer using the AIX operating system routine *copyout*.

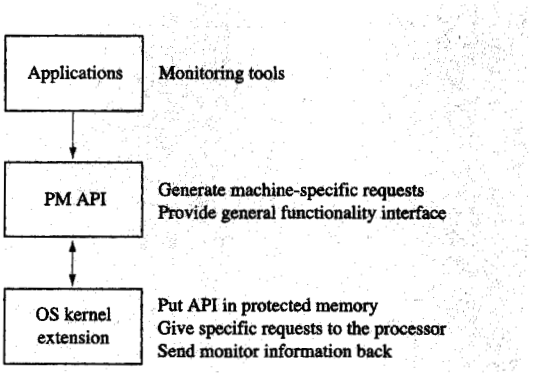

Figure 3

Basic API design.

• *Interface definition considerations*

The interface definitions should follow standards and good programming practices. The interface must map the basic functions required by the applications into a "best fit" provided by the machine-specific support. A method must be provided to identify exactly what was supported on the specific processor. The interface and its updates must be able to support all previously supported functions without requiring the calling code to be changed or even recompiled. The support code must be expandable to allow for new functions to be added. Operating system dependencies should be as isolated (in the code) as possible and handled in such a way as to allow for updates that are operating-system-specific without affecting the support provided for other operating systems. Expected enhancements and expected differences between processors should be isolated in the code so that a given type of support for a new processor is not expected to affect the support previously tested and made available. The interface should facilitate methods which reduce the invasiveness of measurements on the system under test. The interface should provide for counter integrity, which includes exclusive ownership of the counters and the prevention of undetected counter overflows.

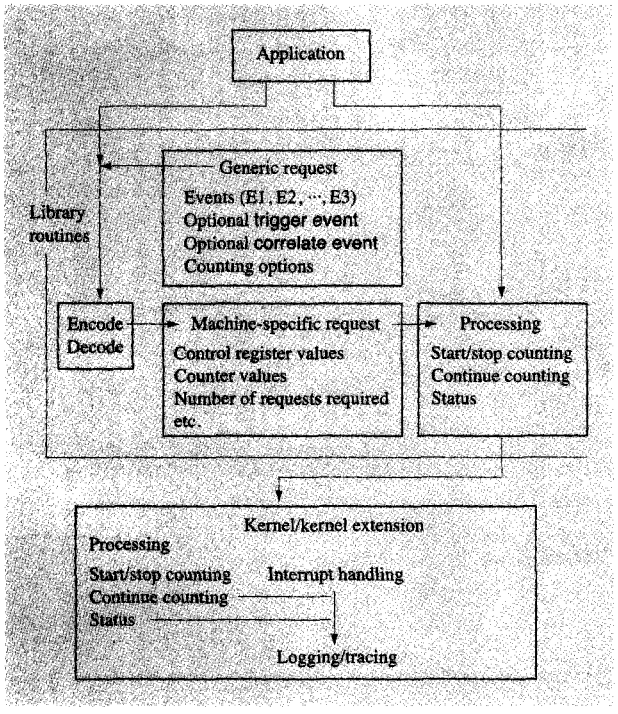

Figure 4

API support functions.

Kernel extension load and unload

A separate utility (PMkexLoad) is supplied to load and unload the kernel extension. Note that this function is available only to a user with special authority on AIX.

Application support routines in the PM API

The PM API includes application-support routines [8], which must be called by application programs using the performance-monitoring functions provided by the kernel extension layer.

Figure 4 shows the PM API library and kernel extension support. This layer of the PM API includes the functions described in the following subsections.

- *Encoding and decoding*

The encoding interface routine [9] maps the caller's specified input, including a generic list of events, into a set of machine-specific controls, which is passed to the routine that initializes processing for the request. Because some of the specified requests may not be available (that is, the required hardware support may not be available on a given processor), the encoding routine must support whatever it can and pass back error information regarding those items it cannot support in hardware.

Since the actual events suported by a given processor are implementation-dependent and the assignment of events to counters is processor-dependent, the number of PM requests required to support a given list of events may vary as a function of the processor. For this reason, there must be a mechanism to tell the caller which events will be counted and on which request. Also, the support for specific parameters may only be approximated; again, there must be a means to identify the exact values supported. Although most of this information is returned by the encoding routine, there are some instances in which "decoding" of the machine-specific information is helpful. The decoding interface [10] can be used when the control registers are read; it takes the machine-specific requests and converts them to the generic format which could be used as input to the encoding routine.

The encoding routine supports the following features:

- A revision field to facilitate the addition of future enhancements without requiring the application using the old support to be modified or even recompiled.
- Flag words and bits within the words that represent counting control functions.
- Flag words and bits within the words that represent exception control functions.
- A separate parameter for the timebase selection function using a generic interface, the standard UNIX** time interface of seconds and nanoseconds. Functions such as timebase selection and configuration support are likely to be specific to the operating system as well as to the machine. Items that are expected to be operating-system-specific are supported in a separate source file for easy repackaging and integration as changes are added to support different operating system environments.
- A separate parameter for the threshold uses the generic interface of processor cycles.
- A list of unique event names each of which is defined with a unique number. The same event name is used on each processor in which the same event is counted.
- A set of parameters allowing the application using this interface to use the name of the events of interest to specify exactly what must be counted. The following specifications are supported: a *trigger event* (an event which must occur some application-specified number of times before additional counting can occur); a *correlate event* (an event which must occur on each PM request); and a list of events to be counted.

- *Process initialization for counting*

The process initialize counting interface (PMprocessInit) supports the initialization of the processing of counting requests which use the encoded, machine-specific information. PMprocessInit verifies that no other process

has initiated counting on the specified processor and uses PMkex to copy the detailed processing control information passed by the caller.

PMprocessInit allows the application to specify detailed processing options, including the processor on which to initialize counting information. PMprocessInit allows the application to specify that counting be started or not started by PMprocessInit. PMprocessInit allows the application to specify the number of passes, that is, the number of times to cycle through all of the events requested in the machine-specific information. This interface provides for an unlimited number of passes. The application must specify operational modes (e.g., what to do on each interrupt). The application may specify that the counters are initialized on each interrupt request or that the counters are not initialized on each interrupt request. The application may specify, for each interrupt, that the same request be issued, the next request be issued, or no requests be issued. The application may specify that it be posted on each interrupt; or when a complete set of events have been processed (i.e., a pass has been completed); or when all the passes have been completed. The application may reissue a PMprocessInit at any time without relinquishing control of the counters.

- *Processing control*

The process control interface (PMprocess) allows the application to specify the processor to which the process control command applies, and a command. The commands allow the application to specify whether to start counting with the same set of events previously counted, to cycle to the next set of events, to stop counting, or to terminate counting. The terminate counting option relinquishes control of the counters, allowing another process to gain control of the counters. The PM API automatically handles abnormal termination cleanup, which allows reuse of the counters by a different process; however, if counting has not been stopped by the application, counting continues until a new PMprocessInit has been issued.

- *Status control*

The status interface routine (PMstatus) provides an interface to read and return the current values of the performance monitor registers and the accumulated 64-bit counts maintained by the PM API support routines. In addition, there is support for returning the values of the PM registers, accumulated 64-bit counts, and other information consistent with the previous interrupt.

PM API application considerations

- *Processor support*

The current external release of the PM API code supports the 604 and the 604e; the current internal release of the PM API supports the 620 and other internal 64-bit processors. The PowerPC processors not supporting the PMs are the PowerPC 601*, the PowerPC 603*, and the PowerPC 603e*. Work is underway to support the P2 and the P2SC.

- *Operating system dependencies*

The AIX version of the PM API requires AIX 4.2, which has added features to enable the support. Since the PM API uses only documented and supported AIX interfaces, it is not expected to require changes in new AIX releases. The PM API is expected to work with AIX reliability, availability, and serviceability (RAS) aids, including trace and diagnostics.

The non-AIX versions of the PM API should support identical library interfaces to encode the machine-specific list from a list of events. However, operating-system-specific code may have to be quite different and may have to be incorporated into the kernel (instead of as a kernel extension) or may have to be modified to use existing operating system interfaces that are already provided in the operating system. There are no IBM plans to port the PM API to a non-AIX operating system.

- *PM API overhead*

As is well known in the physics of elementary particles, the act of measuring can affect the system being measured—Heisenberg's uncertainty principle. Software-based approaches to measurement also tend to affect the system under test. The design of the PM API and the PMs themselves provides for wide flexibility and control over the invasiveness of measurements. There are many different modes of operation and many different applications for measurements.

We now discuss some methodology considerations. All of the work to set up the control information used to actually initiate or change what must be counted should be done prior to running the job to be measured. That is, the utilization of the encoding routine should not be a factor in the overhead related to measuring. The overhead related to initiating counting may be eliminated by judicious use of the MSR PMM bit when this is appropriate. Support was added to AIX 4.2 to support the setting and propagation of the MSR PMM bit in application (problem state). This feature may be used to measure a specific application while it is in problem state. The kernel itself can be compiled to propagate the PMM bit, although the officially shipped AIX 4.2 kernel object code does not provide this function. At any rate, all of the current POWER and PowerPC performance monitors permit counting to be gated by whether or not the MSR PMM bit is set. This allows PM hardware to be initialized for counting before actually starting the process to be monitored. The determination of those processes which

have the PMM bit set can then be done as part of the task of starting those processes. This allows counting to start with no initiation overhead.

The 620 provides a function that allows counting to start automatically at a particular effective address specified by the IABR. Processors that provide support for this function can again provide a no-overhead method for initiating counting. Similarly, the *trigger* function provides a noninvasive way to begin counting in the remaining counter(s) after PMC1 is negative.

If use of the PMM bit is appropriate, counting stops automatically when the job is finished; thus, in this case, the counters can be read after the job has terminated with no additional overhead. With this approach, measurements for jobs that run fairly quickly can be made with no overhead related to the taking of the measurements themselves.

The PM API always accumulates 64 bits worth of counter information for each event being counted in order to avoid the problem of wrapping a counter and losing the information that a counter has wrapped. This feature can be programmed to run automatically, so that overhead is incurred only after a counter has become negative. The overhead associated with this feature is the overhead related to taking the PM interrupt, accumulating the required data, and reinitiating counting as required, about 200:400 instructions. Since this occurs only when a counter has become negative, it is gated by the initial value(s) of the counters and the frequency of update of counters. As an upper bound for this overhead, assuming that the initial value of zero is used, that the number of instructions completed is the fastest incrementing counter, and that the interrupt takes about 400 instructions to execute, we obtain $(400/2^{31}) \times 100\% = 0.0000186\%$, which is clearly negligible.

Another significant mode of operation, called sampling, is performed by taking periodic snapshots of the system. The sampling rate determines the invasiveness of the sample measurements. Assuming that data are written to a log, it takes about an additional hundred instructions to write the data to a log or a total of about five hundred instructions to process a sample. The log itself may be pinned and thus may consume a fixed amount of random access memory (RAM). If the log is pinned, the size of the log determines the total number of samples allowed. The current PowerPC performance monitors allow the sample rate to be selected by the application by allowing a counter to be set to a value which causes the counter to go negative after some number of occurrences of an event (e.g., after one thousand cycles). In addition, these processors provide the capability to identify one of four bits in the timebase and force an interrupt when the selected bit flips. One should note that the increment of the timebase is not architected and is defined to be system-specific. For most PowerPC systems that currently support performance monitoring, the increment of the timebase is a function of some increment of the processor bus speed. The actual bits chosen are bits 47, 51, 55, and 63. The PM API allows the application to specify the desired sample time in seconds and nanoseconds and uses the system configuration data to determine the selection of the closest sample rate. The PM API does not allow bit 63 to be used along with direct interrupts. The PM API provides an option that prevents the system from being overwhelmed by interrupts, if they occur faster than a system-specific rate.

There may be an interest in monitoring more events than there are counters. For repeatable jobs, one may run the entire job with the first set of events that can be counted concurrently, then rerun the job with the next set of events that can be counted concurrently, continuing until the final set of events are counted. Using this approach, one can obtain the total counts for all items that can be counted. For jobs that represent nonrepeatable workloads, this approach is not feasible. The PM API provides a few ways to count more events than can be counted concurrently. The application specifies the events to be counted, an optional *trigger event*, and an optional *correlate event*. The encoding routine constructs the encodings required to count all of the specified events and determines the number of performance monitor requests required to count all of the events. The application may use the data received from the encoding routine to control exactly what gets counted when. The format of the output from the encoding routine is made available to the application and can be modified prior to initiating counting.

The PM API may then be programmed to automatically continue counting the next set of events; that is, on each interrupt, the values for those items being counted are read, and counting is initiated for the next set of items to be counted. The counts are accumulated as appropriate, where the *trigger event* and *correlate event* are accumulated on each request, but the other events are only accumulated once for a full pass (a cycling through all of the events). When a PM exception is signaled, the SIAR and SDAR registers are frozen until exceptions are reenabled. Also, the counters can be programmed to stop counting when the exception is signaled. If the application wishes to control the time at which the next set of events are counted, it can specify to PMprocessInit that no new requests be issued while the PM interrupt is being processed. In this case, the next set of events are counted only when the PMprocess request is issued with the continue to the next set of events option. By using the post option, the application can be posted when an interrupt occurs and can issue a status to obtain the latest sampled data. This approach allows the application to

control the logging of the data and to place the log in nonpinned memory.

The overhead associated with taking an interrupt is significant in typical pipelined superscalar machines. The pipeline is typically drained on entry and exit from the interrupt routine. Also, the interrupt support typically takes up some of the processor's cache when the interrupt code is being executed. These disruptions are also true for each system call supported. It is possible to avoid the cache disruption by putting the interrupt code into noncacheable write-through memory. However, this support is not provided by the PM API, although there is an attempt to minimize the number of system calls and/or interrupts required to provide the desired function.

- *Multiprocessor considerations*

The design of the PM API provides for separate control blocks for each processor. In a uniprocessor environment, the processor number is always forced to zero. When the configuration information identifies more than one processor (say N processors), the process initialization routine uses the AIX bind function (bindprocessor) to force the current process to be bound to each processor, from 0 to $N - 1$. All indexing to the control blocks in an SMP system uses the processor number.

When counting is initialized (PMprocessInit), the application specifies the processor to which it must be bound. The library routine binds the application to the specified processor. The process is left bound to the specified processor after the initialize function is completed.

When a subsequent PMprocess request is issued, the application specifies the processor to which it must be bound. The library routine binds the application to the specified processor. The process is left bound to the specified processor after the process function is completed.

When a PMstatus request is issued, the application specifies the processor to which it must be bound. The library routine binds the application to the specified processor. The process is left bound to the specified processor after the status function is completed.

The PowerPC performance monitors provide for the signaling of exceptions when a chosen bit in the timebase transitions. For all bits other than bit 63, this provides a means to take a system snapshot at intervals far enough apart to allow useful processing to occur. Thus, if the application initiates counting on each processor with the timebase transition bit exception specified, all of the processors will signal the PM exception at the same time. If counting is programmed to stop when the PM exception is signaled, the data read when the PM interrupt is actually taken will reflect the state of the system when the interrupt was signaled. This provides for a cross-sectional view of what is happening on all processors at the same time. On AIX, the application can log and use the logged data to reconstruct the state of the machine on all of the processors. The application can ensure that virtually any relevant data are logged (e.g., the process that is in execution at the time the interrupt is taken). In addition, the application can profile the relative times spent executing specific instructions (available from collecting the SIARs).

Operating systems other than AIX may not support multiple processors. For these systems, the processor parameter should be ignored, as is the case in the current design. Other operating systems that support multiple processors may not provide a bind function. These operating systems may, however, force a process to stay on a specific processor under some specific circumstances such as a kernel or system call. These issues must be explored if and when the code is ported to operating systems other than AIX. Automatic tracing and logging facilities are not currently supported but may be supported in a future version of the PM API. These are areas that must be addressed when the PM API is ported to other operating systems.

- *Reentrancy considerations*

The PM API is designed to be reentrant by processor, where the PM data for each processor are independent from the data maintained for any other processors. The application provides the work areas for the encoding routine and the decoding routine, and all working data areas are automatic or stack variables. The routines in the application library are reentrant by processor. The routines that control the PM counters allow only one process control over the counters for a specific processor by using semaphores. All access to the kernel control blocks occurs while interrupts are inhibited. This effectively provides a lock on all PM requests such as status, continue processing to the next set of events, and interrupt handling.

- *64-bit considerations*

The C code is designed and coded to work correctly with both 32-bit and 64-bit machines. The assembler code is tailored to each machine, with the correct object enabled for execution when the PMkex is loaded as a function of the machine configuration.

I/O space counters

There is some interest in providing additional support to control counting in nonprocessor components such as bridge controllers or memory controllers. The current PM API does not address this issue. However, there is some current work underway to support the P2 and the P2SC.

Conclusion

The 604 PM has been used as a significant evaluation and tuning aid for multiple platforms and individual programs. The existing functionality has been generally regarded as excellent, and its widespread use has ensured that most new PowerPC processors intended for use in servers, workstations, or personal computers will have an on-chip PM. In order to take full advantage of the capabilities in future PowerPC processors, a PM API is being developed. The dissemination of this PM API is expected to facilitate the development of tools that can be used on different processors and systems.

Acknowledgments

The authors wish to express their appreciation to E. Welbon and T. Keller, who helped identify the initial set of events to be counted in the 604. E. Welbon has participated in PM event definition, counter assignment, and architecture definition on many processors. We also would like to thank all of the other talented engineers and software developers across Apple, IBM, and Motorola who contributed to the success of the PowerPC PM strategy and implementation.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intel Corporation or X/Open Co., Ltd.

References

1. IBM Microelectronics Division and Motorola Inc., *PowerPC 604 RISC Microprocessor User's Manual*, Chapter 9: Performance Monitor, 1994, pp. 9-1–9-11.
2. E. Welbon, C. Chan-Nui, D. Shippy, and D. Hicks, "POWER2 Performance Monitor," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1994, pp. 55–62.
3. C. Roth, F. Levine, and E. Welbon, "Performance Monitoring on the PowerPC 604 Microprocessor," *Proceedings of the 1995 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, October 3, 1995, pp. 212–215.
4. C. Roth, F. Levine, E. Welbon, and R. Moore, "Load Miss Performance Analysis Methodology Using the PowerPC™ 604 Performance Monitor for OLTP Workloads," *Proceedings of COMPCON '96, the Forty-First IEEE Computer Society International Conference: Technologies for the Information Superhighway*, February 25–28, 1996, pp. 111–116.
5. C. Roth and F. Levine, "PowerPC Performance Monitor Evolution," *Proceedings of the 1997 IEEE International Performance, Computing and Communications Conference*, February 5–7, 1997, pp. 331–336.
6. H. Dwyer, R. Heisch, F. Levine, and E. Welbon, "Technique for Speculatively Sampling Performance Parameters," *IBM Tech. Disclosure Bull.* **37,** No. 9, 589–592 (September 1994).
7. *The PowerPC™ Architecture: A Specification for a New Family of RISC Processors,* Second Edition, Morgan Kaufman Publishers, Inc., San Francisco, 1994, pp. 351–357, 479–481.
8. F. Levine, "Generic Performance Interface Approach," *IBM Tech. Disclosure Bull.* **39,** No. 8, 65–68 (August 1996).
9. F. Levine and W. Starke, "Generic Performance Monitor Interface Events List Encoding," *IBM Tech. Disclosure Bull.* **39,** No. 8, 251–254 (August 1996).
10. F. Levine, "Decoding of a Specific Performance Monitor Encoded Request to a Generic Format," *IBM Tech. Disclosure Bull.* **39,** No. 8, 169–171 (August 1996).

Received August 8, 1996; accepted for publication June 3, 1997

Frank E. Levine *IBM Microelectronics Division, 11400 Burnet Road, Austin, Texas 78758 (levine@austin.ibm.com).*

Mr. Levine received a B.S. in mathematics from Tufts University in 1970 and an M.S. in mathematics from Purdue University in 1972. He continued taking graduate courses in mathematics and computer science until 1974, when he joined the IBM Federal Systems Division, where he was a lead programmer for various software components for the ground support system for the shuttle at Cape Kennedy, Florida. In 1979, he moved to Austin, Texas, where he was a lead programmer on various software development projects, including DisplayWriter*, DisplayWrite*, and OS/2 EE* Data Base Manager. In 1989, Mr. Levine became a Software Development Program Manager for AIX* RISC System/6000* development projects. In 1992, he joined the PowerPC System Architecture Department and coauthored the book *RISC System/6000 PowerPC System Architecture*, which was published by Morgan Kaufmann Publishers, Inc. In 1995, he started work on the PM API, which he is currently enhancing, with P2 and P2SC support scheduled to be completed in 1997.

Charles P. Roth *Somerset Design Center, IBM Corporation, 11400 Burnet Road, Austin, Texas 78758 (cproth@ibmoto.com).*

Mr. Roth received a B.S. in electrical engineering from Texas A&M University in 1989 and an M.S. in electrical engineering from the University of Texas at Austin in 1991. In 1989 he joined the IBM RISC System/6000 processor design group in Austin, where he worked on the design verification of the POWER and RSC (RISC single-chip) processors. In 1991 he joined the PowerPC microprocessor design team, where he has contributed to the design of the PowerPC 601*, PowerPC 604, and PowerPC 604e microprocessors in the areas of timing analysis, design methodology, performance monitoring instrumentation, and logic design. He is currently working on an undisclosed PowerPC product.