The POWER2 Performance Monitor

E.H. Welbon, C.C. Chan–Nui, D.J. Shippy, and D.A. Hicks

Introduction

The POWER2 Performance Monitor ("monitor"), currently an internal tool, provides detailed hardware measurements necessary to study the hardware/software interaction of workloads executed by the POWER2 processor. This monitor is integrated into the POWER2 processor and is fully software accessible. This monitor can selectively measure specific software processes with minimal disruption.

Performance measures provided include instructions executed, elapsed cycles, counts and delays associated with cache and TLB misses, and utilization of the various execution elements. Such measures can help to efficiently locate and eliminate performance bottlenecks.

The Need For Hardware Measurement

Regardless of the advances in CPU design, improving system performance increasingly depends on a detailed understanding of hardware and software interaction [1]. An effective measurement facility can provide such understanding. An effective measurement facility can also provide the basis for evaluating design decisions of future systems by identifying existing bottlenecks. For example, it is useful to identify opportunities for better CPU execution utilization by improving compiler instruction scheduling. Instances of such opportunities are described in [2, 3]. Development of the compiler can be hastened if information identifying scheduling opportunities is readily available.

Of course, hardware measurement is not the only method of understanding system behavior. Simulation is an effective way to study hardware / software interaction in the earliest phases of the design and implementation of a system [4]. However, if measurable hardware is available, it is clearly preferable to measure rather than simulate.

Drawbacks Inherent to Previous RISC System/6000 Measurement Facilities

In previous RISC System/6000 systems, IBM development teams collected hardware performance information using special external instrumentation, which was adapted to the processor model under consideration.

The external instrumentation approach was possible since the RISC System/6000 multiple chip design provided some access to signals typically internal to a single chip [5]. The implementation also included special signals solely for performance monitoring. These special signals were provided either multiplexed functional pins or by pins dedicated to measurement. Additional logic was typically required to convert the externally available signals into useful performance information.

Subtle differences between the various CPU models required each CPU to have different instrumentation. As a result, instrumentation was not available on all CPU models. Additionally, the difficulty and cost of instrumenting any particular CPU model has further constrained the number of systems instrumented.

Desirable Monitor Characteristics

The availability of a hardware measurement facility can be an asset to understanding hardware / software interaction and thus aid in improving the performance of current and future systems.

A desirable characteristic is the ability to select specific times and specific threads of execution to measure. An analyst studying a numerical analysis application may be interested in particular subroutines or even particular loops [1]. On the other hand, an analyst studying a commercial application may want to be able to exclude I/O wait intervals [6].

A measurement facility is much easier to use if the software can efficiently access and control the measurements of its own execution. This is true whether the instrumentation is internal or external.

Ideally, the monitoring instrumentation should be directly incorporated into the CPU chips and be accessible by software in a simple yet effective manner[1]. An example of such capability is the Cray Y–MP [7] which has software accessible counters. It was desired that POWER2 implementation provide similar capabilities.

Implementation of Monitor

Systems are becoming more complex and highly integrated causing less data to be available for external hardware to intercept and record. One example in particular is the POWER2 processor which is packaged on a Multiple Chip Module (MCM).

The MCM package includes five basic units, each being a distinct chip. These units are: 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 (DCU). Additional details are provided in [8, 9,10,11].

Most of the chip inputs are interconnected on the MCM substrate and not externally accessible. The memory and I/O buses are the only external interfaces to the MCM [11,12]. Consequently, there are few external POWER2 CPU signals suitable for deriving performance data.

Providing the measurement data via software accessible registers can reduce the pin I/O requirements. To this end, the ICU, FPU, SCU, and FXU are each allocated five counters. Similarly, each of the four basic units is provided with a four–bit control field in the Monitor Mode Control Register (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.

As illustrated by Figure 1, the monitor contains twenty–two 32–bit counters which are used to count CPU/Storage performance related events. The MMCR provides monitor control functions. The MMCR, along with two status bits in the POWER Machine State Register (MSR), allows selective measurement of specific threads of execution. The counters and the MMCR are addressable for read and write operations via Programmed I/O (PIO).

Except for the SCU, the selection of the events to count for any unit is independent of the selection of the other three units. A Setting a particular MMCR bit (SGA) will cause the SCU to use all twenty–two counters for SCU events. This feature provides much useful storage access information.

An Instruction Match Register (IMR) provided in the FXU counts the occurrence of specific instructions. By repetitively cycling through all desired instruction codes, a sampling of the instruction execution frequencies can be obtained.

Finally, a bit referred to as the Software Programmable Event (SPE) allows software to create “software events” as desired. The FXU counters can count the number of cycles that the SPE bit is “1” (set by software).

Monitoring States

The POWER architecture defines the system wide Machine State Register (MSR) as part of process state. MSR bit number 29, the The Process Mark (PM) and Problem (PR) bits of the MSR are used to control the state of the monitor.

Figure 2 shows the Monitor Mode Control Register (MMCR) which provides control modes that take advantage of the MSR PM and PR bits. Since the MSR is part of process state and is saved and restored when processes pause and resume execution, the MSR PM bit can be used to selectively qualify processes for monitoring with low overhead. Thus the MMCR and MSR together can efficiently control the state of the Monitor.

Selection of Observable Events

The previous RISC System/6000 measurement facilities rely on externally accessible signals that external hardware converts into performance information. The POWER2 monitor includes hardware in each unit that directly detects the performance information.

1. Generally, the signals necessary to detect the required event will be available within the chip. Signal availability caused by pin restrictions tend to prohibit this on external instrumentation.

Therefore, the selection of the POWER2 performance events is based on what the users of the performance community wanted as opposed to what could be observed given the available signals. As a consequence of the monitor's organization, the chip provides a wealth of event information. This paper is not intended to serve as an implementation guide; the following four sections provide a summary of these events.

FXU Observable Events

The FXU provides cycle counts for categories such as branch delays, data cache misses, floating–point interactions, TLB reloads[2], and the timing of multiple cycle instructions. The FXU can count the number of instructions that it executes. Other counters allow monitoring the utilization of the various fixed–point execution elements.

A special set of counts identify cases where opportunity exists for performance improvement via alterations of the scheduling of the instruction streams being executed. A closely related set of counts identify the *kind* of scheduling opportunities; these counts categorize the causes of FXU idle cycles. Thus, the type of opportunity, as well as an upper bound estimate of its magnitude, can be identified.

2. In the POWER and POWER2 implementations, storage accesses that miss in either the data or instruction TLB are satisfied by the FXU in hardware and, barring page faults, are transparent to software.

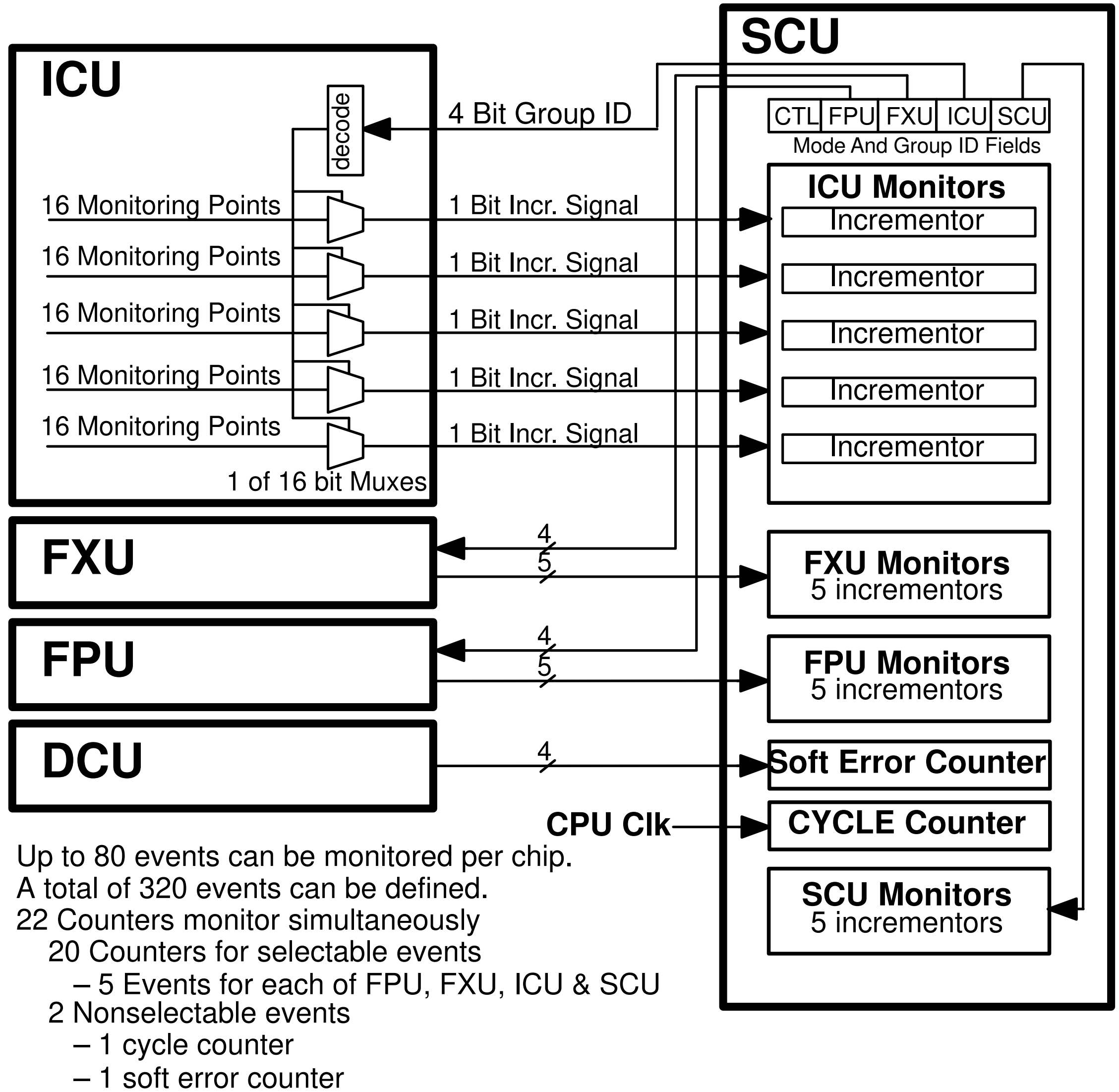

Figure 1 Organization Of Monitor Hardware

ICU Observable Events

The ICU provides counts of instruction cache and TLB misses, the number of instructions fetched, dispatched, and executed, and the number of interrupts acknowledged. Also provided is a breakdown of the classes of instructions executed (principally branch instructions), and a set of counts identifying the sources of dispatch and branch delay.

FPU Observable Events

The FPU provides counts that give many details on conditions causing floating–point hold off cycles, execution element utilizations, the number of instructions executed, and IEEE special number (e.g. infinity, NaN, etc.) processing delays. In addition, the FPU provides counts of the classes of instructions executed, counts of cycles incurring register renaming stalls, and counts of various queue related stall cases.

SCU Observable Events

The SCU counts the number and type of storage operation requests that are presented to it, measures of latency for such requests, and measures of DMA, memory bus, and SIO bus activity. In addition to housing the selectable counters, the SCU contains two dedicated counters: a CPU cycle counter and a correctable memory error counter.

Storage performance is critical in RISC machines [13]. To most effectively measure storage–caused delays, a special mode was defined for the SCU. Ordinarily, the FXU, ICU, FPU, and SCU can operate only five counters each. However, Figure 2 describes

MONITOR MODE CONTROL REGISTER (MMCR)

DIS	DP	DU	DMS	DMR	SGA	//	FPU	FXU	ICU	SCU
0	1	2	3	4	5	6	16	20	24	28

This register is set to zero on power up. Reading this register does not change its content. Bits 6 though 15 are reserved, and always read as zeroes. 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 a one, 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 one, causes the counters to stop counting when MSR(PR) is a zero. The counter values are not changed, only suspended.

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

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

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

SGA: SCU Gets All: This bit, when set to a one allows the SCU to control all twenty two counters i.e., the SCU gets all of the counters. This bit over rides the FPU, FXU, ICU and SCU source event set selection.

FPU: MMCR[16:19], 4–bit code selecting FPU source event set.

FXU: MMCR[20:23], 4–bit code selecting FXU source event set.

ICU: MMCR[24:27], 4–bit code selecting ICU source event set.

SCU: MMCR[28:31], 4–bit Code selecting SCU source event set.

Figure 2 Monitor Mode Control Register Format a special case involving the SGA bit of the MMCR that allows the SCU to operate twenty–one of the twenty–two counters with strictly storage control related counts. This mode provides a detailed view of the storage and I/O activity.

Basic Monitoring Considerations

In general the monitor is a serially reusable resource and must be managed to ensure that it is utilized in a reliable manner. If the counters are managed by the operating system at a user's request, multiplexing the monitor function across multiple processes is possible. In this case, the counters are treated as part of process state and are saved and restored to allow each process to use the monitor as though it is private to each process.

Because the monitor cannot count every event simultaneously, alternating the events to be counted can provide more data. This technique is valid when monitoring a steady state or cyclic environment. On a periodic basis, one could cycle the values of FPU/FXU/ICU/SCU counter event selection fields through the full range of values. This technique provides a sample of all measurable events. Such a procedure was used to obtain the data for the results of [6] and is discussed in the "Monitor Data Reduction Tools" section of this paper.

Selection Of Code to be Monitored

A key feature of the monitor is the ability to automatically focus on supervisory activities or particular processes with no overhead. Conversely, particular processes or supervisor activity can be excluded from the measurement process. Some of the possibilities will be described in the following four sections.

Monitoring Specific Blocks Of Code Within Processes and Programs

An analyst studying a numerical analysis application might be interested in particular subroutines or even particular loops [1]. Such measurements may be made by placing a call to enable monitoring at the desired entry point and corresponding call to extract values at the appropriate exit point. Although the calls to enable and disable monitoring will incur some overhead, this overhead will not markedly affect the measurements made on the code segment being analyzed. This technique is further discussed in the "Monitor Command Line Tools" section of this paper.

Monitoring One Process

Many performance studies focus on workloads consisting of a single process. In such a case, the MMCR and the MSR control functions described in Figure 2 can be used to focus performance monitoring to just a single process with no software overhead beyond setup. This is discussed further in the "Monitor Command Line Tools" section of this paper.

The benchmark in question will have both MSR(PM) and MMCR(DMR) set to one. All other processes executing would have their respective MSR(PM) bits set to zero. As a consequence, only the benchmark process with MSR(PM)=1 will be able to alter the performance counters. Thus, the performance counters would not have to be saved or restored, eliminating most of the control overhead.

Since the counters will "roll over" roughly every 65 seconds (depending on the processor clock speed), it will be necessary for the operating system to schedule an event that will save the performance counters. Also, the monitor control software must supply certain managerial functions for the monitor, since reading and writing the MMCR and the MSR are privileged operations.

Monitoring A Set Of Processes

It is possible to monitor the cumulative effects of any set[3] of the executing processes using the above technique. 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 bit set to one.

In this technique a fixed set of parameters is measured for the processes of the set. Each time a process from the set to be monitored commences execution, that process's MSR will be loaded with a PM bit value which will enable monitoring. Note that there will be no need to save or restore the monitor counters or MMCR in this case. The data values

3. A simple example of such a set would be an AIX "process group". In this case, each process will inherit the process state of its parent process; if the parent process has the requisite MMCR and MSR bit settings, then these settings will be automatically propagated to the parent process's children.

produced by the counters is treated in the same way as the data values produced by a single process.

Monitoring All Processes Using A Log
To obtain a trace of counts for all process and operating system activity, an algorithm similar to that used by the *aixtrace* [14] has been 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 by [14].

Because all processes are monitored, the MSR is not needed to select specific processes. Note that the monitor counters in this case are considered part of a process's state just as its general purpose registers are.

At desired intervals, the counter values are written to the *aixtrace* trace log. Each entry consists of the process id, the MMCR, the counter values, and the real time clock at the time that the counters were read.

The implementation of this procedure is discussed in the "System Level Software" section of this paper.

Monitor Support Software

In the balance of the paper we discuss the monitor support software. This software provides performance information to the analyst for both system level and process specific measurement strategies. The next section, "System Level Monitor Software", describes a measurement strategy which provides system level data suitable for overall system analysis and visualization software [15].

The section entitled "Monitor Command Line Tools" describes software that efficiently provides process level measurement. Finally, the "Monitor Data Reduction Tools" section discusses the software used to convert the raw monitor data into useful performance information. This software has been successfully used to analyze several benchmarks including TPC as described in [6].

In all of the following methods, the monitor support software is implemented as an AIX kernel extension.

System Level Monitor Software
This section describes the manner in which the monitor interfaces with the *aixtrace* facility [14]. This facility provides a detailed view of all processes that execute and, in particular, captures the interaction between system and application software. This feature is constructed around a special trace subroutine call provided by the AIX kernel.

Trace calls which request the capture of information, are embedded at desired locations in the software to be monitored. These calls are referred to as “hooks”. Each distinct “hook” has a unique identifier, the “hook” ID, which distinguishes between the various “hooks”.

These “hooks” may be system or user defined and may be disabled and enabled as desired. Each “hook” is designed to pass “hook” specific data. In particular, the monitor counter data can be inserted into the *aixtrace* log as an *aixtrace* record every *n* software clock ticks (or when the process dispatched is different from the current process).

The monitor kernel extension intercepts trace calls. The monitor examines the arguments of the “hook” and determines if it is an “appropriate” time to record the monitor data. If so, the kernel extension calls the trace routine on its own behalf, inserting the counter data into the trace.

This approach is flexible since the kernel extension allows the user to specify a list of which “hooks” will cause the monitor counters to be recorded. This allows report generators that postprocess the trace to produce per process performance statistics. Also, as is characteristic of kernel extensions, enhancements to the monitor facility may be added without rebuilding the kernel.

The main drawback of this method is that the *aixtrace* facility and the appropriate kernel extension must be enabled. This means that depending on the number of “hooks” enabled, data gathering is slightly intrusive (in the neighborhood of 10%).

Monitor data for visualization software such as described in [15] is provided through the *aixtrace* facility. This allows the visualization software to correlate the data provided by the performance monitor with the normal trace data to provide more informative displays.

A more efficient way of implementing this functionality involves calling the monitor kernel extension directly rather than via “hook” invocations. The Command Line Tools discussed in the next section address this feature.

Monitor Command Line Tools

Monitor command line tools provide a simple efficient way to acquire the performance monitor counters. These command line programs and system calls 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 (under 65 seconds with a 66 MHz processor), a user can reset the 32–bit counters, run the program, and acquire the counter values, without the possibility of overflow. Thus, the program can be monitored without intrusion.

For longer programs it is necessary to periodically save the counters to prevent them from overflowing. One of the tools operates somewhat like *iostat* [14] and writes the counters at a frequency specified by the user. Using this program or an analogous one, the amount of intrusion is roughly equivalent to a context switch each time the counters are recorded.

To measure specific sequences of code within a program (e.g. loops or subroutine calls), the user can enable and access the monitor with calls as shown in Figure 3.required

Monitor Data Reduction Tools

The large amount of data collected by the monitor introduces a need for data reduction

```
main() {

   /* code NOT monitored */

   ph_init(MMCR,IMR);
   ph_start();

  /* code to be monitored */

   ph_stop();
   ph_read(COUNTERS);
   print(COUNTERS);

   /* code NOT monitored */
}
```

Figure 3 Coding Example

tools to help the user interpret the data more efficiently. To reduce the data generated by the monitor into useful information, two methods have been implemented. In both cases the monitor data is not dynamically analyzed; it is captured for subsequent processing.

The first method reduces the captured monitor data into sets of measures useful to analysts. Examples of these measures include total instructions executed, cache and TLB miss rates and delays, and utilizations of the various CPU internal elements. To easily generate these performance measures, a set of post processors was developed.

Computing a wide range of interesting derived values is possible. Rather than enumerate the complete set of such values, the post processor accepts user–specified algorithms to determine how to arrive at a desired set of results. Based on the algorithms, the post processor reduces the input data into performance information, complete with statistics.

Based on the recorded MMCR values, the post processor automatically selects the algorithm appropriate for each given measurement. Since the command line tools can cycle the MMCR event selection fields though the range of permitted values, the full set of events provided in hardware can be obtained easily.

The second data reduction method relies on the IBM Research ”PV” Program Visualization Prototype [15]. This visualization tool processes the *aixtrace* log file with a filter that converts the log file data into a format suitable for visualization.

Because the *aixtrace* log file contains significant amounts of operating system data in addition to the monitor data, the visualization tool 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. Application of this feature helped provide the performance information discussed in [6].

Summary

The POWER2 Performance Monitor provides hardware measures that are important to software developers tuning the operating system and application software as well as system designers responsible for the development of new systems. The measures characterize storage system performance and compiler instruction scheduling opportunity, both critical to good system performance [6].

Acknowledgements

We would like to thank the members of the AIX and System Performance groups, and the Engineering Center for their help in developing the monitor. A partial list includes Jama Barreh, Michael Fortin, Maurice Franklin, Richard Fry, Robert Golla, Randy Heisch, John Iacoletti, Doug Kimelman, Peter Markstein, John O'Quin, Bryan Rosenburg, Sohel Saiyed, Ed Silha, and Larry Thatcher. This paper was considerably improved by the comments of John Reysa, Donald Tsang, Charlie Ward, and Steve White. Any inadequacies are the responsibility of the authors.

References

1. S.R. Saiyed, M.J. O'Connor and M.T. Franklin, "POWER2 CPU–Intensive Workload Performance," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
2. H.S. Warren Jr.,"Instruction Scheduling for the IBM RISC System/6000 processor," *IBM J. Res. Develop.,* Vol. 34, No. 1, 1990, pp.85–92.
3. M.C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM J. Res. Develop.*, Vol.34, No.1, 1990, pp. 93–97.
4. Syed Z. Pasha and E.H. Welbon, "Performance–Directed Design Guidance Using Simulation," *IBM RISC System/6000 Technology*, Order Number SA23–2619, IBM Corporation, 1990, pp. 78–85.
5. G.F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM J. Res. Develop.*, Vol. 34, No. 1, 1990, pp.37–58.
6. M.T. Franklin, W.P. Alexander, R. Jauhari, A.M.G. Maynard, and B.R. Olszewski, "POWER2 Commercial Workload Performance," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
7. *Cray X–MP Computer Systems 4–Processor Mainframe Reference Manual*, Order Number HR–0097, pp. C1–C4.
8. L.E. Thatcher, T.W. Griffith Jr. and D.J. Shippy, "POWER2 Fixed–Point, Data Cache, and Storage Control Units," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
9. Baba Arimilli, J.I. Barreh, Robert Golla, and Paul Jordan, "POWER2 Instruction Cache Unit," *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
10. T.N. Hicks, R.E. Fry, and P.E. Harvey, "POWER2 Floating–Point Unit: Architecture and Implementation", *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
11. S.W. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000", *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
12. R.S. Reddy, J.L. Huerta, and H.C. Nguyen, "POWER2 Packaging Technology", *IBM RISC System/6000 Technology: Volume II,* IBM Corporation, 1993, pp.??.
13. John Cocke and V. Markstein, "The Evolution of RISC Technology at IBM", *IBM J. Res. Develop.*, Vol.34, No. 1, 1990, pp.4–11.
14. *IBM AIX Version 3.2 for RISC System/6000, Performance Monitoring and Tuning Guide*, Order Number SC23–2365, IBM Corporation.
15. D.N. Kimelman and T.A. Ngo, "The RP3 Program Visualization Environment", *IBM Journal of Research and Develop.*, Vol.35, 1991, pp.635–651 .