# A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware


Shirley V. Moore

Innovative Computing Laboratory, University of Tennessee
Knoxville, TN 37996-3450 USA
`shirley@cs.utk.edu`



**University of Tennessee Computer Science Department**
**Technical Report UT-CS-02-478**



**Abstract.** Performance monitoring hardware is available on most modern microprocessors in the form of hardware counters and other registers that record data about processor events. This hardware may be used in counting mode, in which aggregate events counts are accumulated, and/or in sampling mode, in which time-based or event-based sampling is used to collect profiling data. This paper discusses uses of these two modes and considers the issues of efficiency and accuracy raised by each. Implications for the PAPI cross-platform hardware counter interface are also discussed.


## 1 Introduction

Most modern microprocessors provide hardware support for collecting performance data [2]. Performance monitoring hardware usually consists of a set of registers that record data about the processor's function. These registers range from simple event counters to more sophisticated hardware for recording data such as data and instruction addresses for an event, and pipeline or memory latencies for an instruction. The performance monitoring registers are usually accompanied by a set of control registers that allow the user to configure and control the performance monitoring hardware. Many platforms provide hardware and operating system support for delivering an interrupt to performance monitoring software when a counter overflows a specified threshold.

Hardware performance monitors are used in one of two modes: 1) counting mode to collect aggregate counts of event occurrences, or 2) statistical sampling mode to collect profiling data based on counter overflows. Both modes have their uses in performance modeling, analysis, and tuning, and in feedback-directed compiler optimization. In some cases, one mode is required or preferred over the other. Platforms vary in their hardware and operating system support for the two modes. Some platforms, such as IBM AIX Power3, primarily support counting mode. Some, such as the Compaq Alpha, primarily support profiling mode. Others, such as the IA-64, support both modes about equally well. Either mode may be derived from the other. For example, even on platforms that do not support hardware interrupt on counter overflow, timer interrupts can be used to periodically check for counter overflow and thereby implement statistical sampling in software. Or, if the platform primarily supports statistical profiling, event counts can be estimated by aggregating profiling data. However, the degree of platform support for a particular mode can greatly affect the accuracy of that mode.

Although aggregate event counts are sometimes referred to as "exact counts", and profiling is statistical in nature, sources of error exist for both modes. As in any physical system, the act of measuring perturbs the phenomenon being measured. The counter interfaces necessarily introduce overhead in the form of extra instructions, including system calls, and the interfaces cause cache pollution that can change the cache and memory behavior of the monitored application. The cost of processing counter overflow interrupts can be a significant source of overhead in sampling-based profiling. Furthermore, a lack of hardware support for precisely identifying an event's address may result in incorrect attribution of events to instructions on modern super-scalar, out-of-order processors, thereby making profiling data inaccurate.

Because of the wide range of performance monitoring hardware available on different processors and the different platform-dependent interfaces for accessing this hardware, the PAPI project was started with the goal of providing a standard cross-platform interface for accessing hardware performance counters [1]. For a related project, see [10]. PAPI proposes a standard set of library routines for accessing the counters as well as a standard set of events to be measured. The library interface consists of a high-level and a low-level interface. The high-level interface provides a simple set of routines for starting, reading, and stopping the counters for a specified list of events. The low-level interface allows the user to manage events in *EventSets* and provides the more sophisticated functionality of user callbacks on counter overflow and SVR4-compatible statistical profiling. Reference implementations of PAPI are available for a number of platforms (e.g., Cray T3E, SGI IRIX, IBM AIX Power, Sun Ultrasparc Solaris, Linux/x86, and Linux/IA-64). The implementation for a given platform attempts to map as many of the standard PAPI events as possible to the available platform-specific events. The implementation also attempts to use available hardware and operating system support – e.g., for counter multiplexing, interrupt on counter overflow, and statistical profiling.

Through interaction with the high performance computing community, the PAPI developers have chosen a set of hardware events deemed relevant and useful in tuning application performance. Because modern microprocessors have multiple levels in the memory hierarchy, optimizations that improve memory utilization can have major effects on performance. PAPI provides a large number of events having to do with the memory hierarchy – e.g., cache misses for different levels of the memory hierarchy, and TLB (translation lookaside buffer) misses. PAPI metrics include counts of the various types of instructions completed, including integer, floating point, load, and store instructions. Also included are events for measuring how heavily different functional units are being used, and for detecting when and why pipeline stalls are occurring. The application programmer may be able to use pipeline performance data, together with compiler output files, to restructure application code so as to allow the compiler to do a better job of software pipelining. Another useful measure is the number of mispredicted branches. A high number for this event indicates that something is wrong with the compiler options or that something is unusual about the algorithm. See [1] for a more detailed discussion of uses of PAPI metrics for application performance tuning.

The remainder of the paper is organized as follows: Section 2 discusses usage models of hardware performance monitoring. Section 3 discusses accuracy issues. Section 4 explores implications for the PAPI interface. Section 5 gives conclusions and describes plans for future work.

## 2 Usage Models

There are basically two models of using performance monitoring hardware:

- the *counting* model, for obtaining aggregate counts of occurrences of specific events, and
- the *sampling* model, for determining the frequencies of event occurrences produced by program locations at the function, basic block, and/or instruction levels.

The first step in performance analysis is to measure the aggregate performance characteristics of the application or system under study [8, 13]. Aggregate event counts are determined by reading hardware event counters before and after the workload is run. Events of interest include cycle and instruction counts, cache and memory access at different levels of the memory hierarchy, branch mispredictions, and cache coherence events. Event rates, such as completed instructions per cycle, cache miss rates, and branch mispredictions rates, can be calculated by dividing counts by the elapsed time.

The profiling model can be used by application developers, optimizing compilers and linkers, and run-time systems to relate performance problems to program locations. With adequate support for symbolic program information, application developers can use profiling data to identify performance bottlenecks in terms of the original source code. Application performance analysis tools can use profiling data to identify performance critical functions and basic blocks. Compilers can use profiling data in a feedback loop to optimize instruction schedules.

For example, on the SGI Origin the `perfex` and `ssrun` utilities are available for analyzing application performance [13]. `perfex` can be used to run a program and report either "exact" counts of any two selected events for the R10000 (or R12000) hardware event counters, or to time-multiplex all 32 countable events and report extrapolated totals. This data is useful for identifying what performance problems exist (e.g., poor cache behavior identified by large number of cache misses). `ssrun` can be used to run the program in sampling mode in order to locate where in the program the performance problems are occurring.

Tools such as vprof [15] and HPCView [7] make use of profiling data provided by sampling mode to analyze application performance. `vprof` provides routines to collect statistical profiling information, using either time-based or counter-based sampling (using PAPI), as well as both command-line and graphical tools for analyzing execution profiles on Linux/Intel machines. HPCView uses data gathered using `ssrun` on SGI R10K/R12K systems, or `uprofile` on Compaq Alpha Tru64 Unix systems, followed by "`prof` -lines", and correlates this data with program source code in a browsable display.

Aggregate counts are frequently used in performance modeling to parameterize the models. For examples, the methodology described in [14] generates

- a *machine signature* which is a characterization of the rate at which a machine carries out fundamental operations independent of any particular application, and
- an *application profile* which is a detailed summary of the fundamental operations carried out by the application independent of any particular machine.

The method applies an algebraic mapping of an application profile onto a machine signature to arrive at a performance prediction. A benchmark called MAPS (Memory Access Pattern Signature) measures the rate at which a single processor can sustain rates of loads and stores depending on the size of the problem and the access pattern. Hardware performance counters are used to measure cache hit rates of routines and loops in an application which are then mapped onto the MAPS curve. Similarly, the "back-of-the-envelope" performance prediction tool described in [12] makes use of aggregate event counts to construct hardware and software profiles. A given hardware and software profile pair are then combined in algebraic equations to produce performance predictions.

## 3 Accuracy Issues

Previous work has shown that hardware counter data may not be accurate, especially when the granularity of the measured code is insufficient to ensure that the overhead introduced by counter interfaces does not dominate the event counts [9]. The analysis in [9] made use of three microbenchmarks to study eight MIPS R12000 events. For each of the microbenchmarks, predicted events counts were compared with the measured counts for both the `perfex` and `libperfex` interfaces. For the loop benchmark, the counts measured using `libperfex` were within 5 percent of the predicted counts for four events when the number of loop iterations was at least 250. However, to get the counts generated using `perfex` within 5 percent of the predicted counts, the number of loop iterations had to be at least 100,000. To relate this work to the PAPI interface on various platforms, we measured the overheads for starting/stopping and for reading the counters in terms of processor cycles. These results, as well as overheads we measured for libperfex, are shown in the table below.

Since the conclusion in [9] is that, given the overhead of the counter interface on a platform, the accuracy of counter data depends heavily on the granularity

| | Linux/x86 | Linux/IA-64 | Cray T3E | IBM Power3 | MIPS R12K |
|---|---|---|---|---|---|
| PAPI start/stop (cycles/call pr) | 3524 | 22115 | 3325 | 14199 | 24850 |
| PAPI read (cycles/call) | 1299 | 6526 | 1514 | 3126 | 9810 |
| libperfex start/read (cycles/call pr) | | | | | 5842 |

of the measured code, we would expect the number of iterations required to get within 5 percent error using PAPI to be close to the 250 required for libperfex on the SGI MIPS R12K, with the exception of the PAPI SGI MIPS R12K interface, which appears to be less efficient that libperfex.

Many profiling tools rely on gathering samples of the program counter value (PC) on a periodic counter overflow interrupt. Ideally, this method should produce a PC sample histogram where the value for each instruction address is proportional to the total number of events caused by that instruction. On modern out-of-order processors, however, it is often difficult or impossible to identify the exact instruction that caused the event.

The Compaq ProfileMe approach addresses the problem of accurately attributing events to instructions by sampling instructions rather than events[5, 6]. An instruction is chosen to be profiled whenever the instruction counter overflows a specified random threshold. As a profiled instruction executes, information is recorded including the instruction's PC, the number of cycles spent in each pipeline stage, whether the instruction caused I-cache or D-cache misses, the effective address of a memory operand or branch target, and whether the instruction completed or if not, why it aborted. By aggregating samples from repeated executions of the same instruction, various metrics can be estimated for each instruction. Information about individual instructions can be aggregated to summarize the behavior of larger units of code. The ProfileMe hardware also supports *paired sampling*, which permits the sampling of multiple instructions that may be in flight concurrently and provides information for analyzing interactions between instructions.

To precisely identify an event's address, the Itanium processor provides a set of *event address registers* (EARs) that record the instruction and data addresses of data cache misses for loads, or the instruction and data addresses of data TLB misses [8]. To use EARs for statistical sampling, one configures a performance counter to count an event such as data cache misses or retired instructions and specifies an overflow threshold. The data cache EAR repeatedly captures the instruction and data address of actual data cache load misses. When the counter overflows, an interrupt is delivered to the monitoring software. The EAR indicates whether or not a qualified event was captured, and if so, the observed event addresses are collected by the software which then rewrites the performance counter with a new overflow threshold. The detection of data cache load misses requires a load instruction to be tracked during multiple clock cycles from instruction issue to cache miss occurrence. Since multiple loads may be in flight simultaneously and the data cache miss EAR can only trace a single load at a time, the mechanism will not always capture all data cache misses. The processor randomizes the choice of which load instructions are tracked to prevent the same data cache load miss in a regular sequence from always being captured, and the accuracy is considered to be sufficient for statistical sampling.

Sampling by definition introduces statistical error. Samples for individual instructions are used to estimate instruction-level event frequencies by multiplying the number of sampled event occurrences by the inverse of the sampling rate. For example, assume an average sampling rate of one sample every $S$ fetched instructions. Let $k$ be the number of samples having a property $P$. The actual number of fetched instructions with property $P$ may be estimated as $kS$. Let $N$ be the total number of instructions, and let $f$ be the fraction of those having property $P$. Then the expected value of $kS$ is $fN$, and $kS$ will converge to $fN$ as the number of samples increases. However, the rate of convergence may vary depending on the frequency of property $P$ and the coefficient of variation of $kS$. Infrequent events or long sampling intervals will require longer runs to get enough samples for accurate estimates.

## 4 Implications for PAPI

The PAPI cross-platform interface to hardware performance counters supports both counting and sampling modes. For counting mode, routines are provided in both the high-level and low-level interfaces for starting, stopping, and reading the counters. For sampling mode, routines are provided in the low-level interface for setting up an interrupt handler for counter overflow and for generating SVR4-compatible profiling data with sampling based on any counter event. Beneath the platform-independent high-level and low-level interfaces lies a platform-dependent substrate that implements platform-dependent access to the counters. To port PAPI to a new platform, only the substrate needs to be reimplemented. Since platform dependencies are isolated in the substrate, changes in the implementation at this level do not affect the platform-independent interfaces, other than making the operations more efficient or providing platform-independent features that had not previously been available on that platform.

The PAPI substrate implementations attempt to use the most efficient and accurate facilities available for native access to the counters. Furthermore, PAPI attempts to use hardware support for counter overflow interrupts and profiling where available. Where hardware and operating system support for counter overflow interrupts and profiling is not available, PAPI implements these features in software on top of hardware support for counting mode. However, the converse has not been attempted – i.e., on platforms such as the Compaq Alpha Tru64 that primarily supports sampling mode, PAPI does not currently implement counting mode in software on top of sampling mode. Although such an implementation is theoretically possible, it raises questions about the accuracy of the resulting event counts since they would be estimated from instruction samples rather than each event being counted by the hardware.

Although the PAPI interface supports profiling based on PC sampling (or, where available, on hardware support for identifying the instruction address for an event), it does not provide access to other information that may be available for the instruction that caused an event, such as data operand addresses or latency information. Nor does PAPI support qualification by opcode or by instruction or data addresses in either counting or sampling modes, although such qualification is available on some platforms such as the IA-64. For example, the Itanium processor provides a way to determine the address associated with a cache miss. It also provides a way to limit cache miss counting to misses associated with a user-determined area of memory. These facilities could enable presentation of data about cache behavior in terms of program data structures at the source code level. Work reported in [3] has shown that such information can be extremely useful in identifying performance bottlenecks caused by bad cache behavior. In [3], the data were obtained through use of a cache simulator which runs considerably slower than the original application (e.g., by a couple of orders of magnitude) and does not model details such as pipelining and multiple instruction issue. Through use of appropriate hardware support (e.g., as on the Itanium), similar data could be obtained more accurately and efficiently.

Although the PAPI library itself does not have any functionality for estimating or compensating for errors, some utility programs have been provided with the PAPI distribution that make some initial attempts. The `cost` utility measures the overheads in both the number of additional instructions and the number of machine cycles to executing the `PAPI_start/PAPI_stop` call pair and the `PAPI_read` call. The `calibrate` utility runs a benchmark for which the number of floating point operations is known and reports the output of the `PAPI_flops` call compared with the known number. Error measurement and compensation may be most appropriately implemented at the tool layer rather than at the library layer. However, the PAPI library may be able to provide mechanisms to enable tools to collect the necessary data.

## 5 Conclusions and Future Work

It is clear that both counting and sampling modes of using hardware performance monitors have their uses and that both should be supported on as many platforms as possible. However, more work is needed to determine which features are most desirable to support in a cross-platform interface and to study accuracy issues related to both models.

Because PAPI presents a portable interface to hardware counters, PAPI is a good vehicle for exploring usability and accuracy issues. PAPI is a project of the Parallel Tools Consortium [11], which provides a forum for discussion and standardization of functionality that may be added in the future. Because of lack of experience with newly available features such as event qualification and data address recording, it seems desirable to experiment with these features before attempting to standardize interfaces to them. The low-level PAPI interface has a routine (`PAPI_add_pevent`) for implementing programmable events by passing a pointer to a control block to the underlying PAPI substrate for that platform. The routine could be used, for example, to set up event qualification on the Itanium. A corresponding low-level routine (`PAPI_read_pevent`) has been added to the developmental version of PAPI to allow arbitrary information to be collected. We plan to use programmable events to experiment with new hardware performance monitoring features that are becoming available, with the goal of later proposing standard interfaces to the most useful features. The `PAPI_profil` call simply generates PC histogram data of where in the program overflows of a specified hardware counter occur. We plan to implement a modified version of this routine that will take a control block as an additional input and allow return of arbitrary information, so as to enable collection of additional information about the sampled instruction (e.g., data addresses, pipeline or memory access latencies). The goal will again be future standardization of the most useful profiling features.

Through the use of microbenchmarks as in [9], we plan to evaluate the accuracy of counter values obtained by the PAPI interface on all supported platforms. Where possible, we will provide calibration utilities that attempt to compensate for measurement errors. We also plan to do statistical studies of the accuracy and convergence rates of profiling data on different platforms, and to investigate the feasibility and accuracy of implementing counting mode in software on top of hardware-supported profiling mode.

For the PAPI software and supporting documentation, as well as pointers to reference materials and mailing lists for discussion of issues described in this paper, see the PAPI web site at http://icl.cs.utk.edu/papi/.

## References


1. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications **14:3** (Fall 2000) 189–204.
2. Browne, S., Dongarra, J., Garner, N. London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Optimization Using Hardware Counters. SC'2000. Dallas, Texas. November,2000.
3. Buck, B., Hollingsworth, J.K.: Using Hardware Performance Monitors to Isolate Memory Bottlenecks. SC'2000. Dallas, Texas. November, 2000.
4. Burger, D., Austin, T. M.: The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report 1942. June, 1997. http://www.cs.wisc.edu/~mscalar/simplescalar.html
5. Dean, J., Hicks, J., Waldspurger, C. A., Weihl, W. E., Chrysos, G.: *ProfileMe:* Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. 30th Symposium on Microarchitecture (Micro-30). December, 1997.
6. Dean, J., Waldspurger, C. A., Weihl, W. E.: Transparent, Low-Overhead Profiling on Modern Processors. Workshop on Profile and Feedback-Directed Compilation. Paris, France. October, 1998.
7. HPCView: http://www.cs.rice.edu/~dsystem/hpcview/
8. Intel IA-64 Architecture Software Developer's Manual, Volume 4: Itanium Processor Programmer's Guide. Intel, July 2000. http://developer.intel.com/

9. Korn, W., Teller, P., Castillo, G.: Just how accurate are performance counters? 20th IEEE International Performance, Computing, and Communications Conference. Phoenix, Arizona. April, 2001.
10. PCL - the Performance Counter Library: http://www.kfa-juelich.de/zam/PCL/
11. Parallel Tools Consortium: http://www.ptools.org/
12. Pressel, D.: Envelope: A New Approach to Performance Prediction. Department of Defense HPC Users Group Conference. Biloxi, Mississippi. June, 2001.
13. Origin 2000 and Onyx2 Performance Tuning and Optimization Guide. SGI Document number 007-3430-003. July, 2001. http://techpubs.sgi.com/
14. Snavely, A., Wolter, N., Carrington, L.: Modeling Application Performance by Convolving Machine Signatures with Application Profiles. IEEE 4th Annual Workshop on Workload Characterization. Austin, Texas. December, 2001.
15. The Visual Profiler: http://aros.ca.sandia.gov/~cljanss/perf/vprof/