

Accuracy of Performance Monitoring Hardware

Michael E. Maxwell, Patricia J. Teller, and Leonardo Salayandia
University of Texas-EI Paso
(mmaxwell, pteller, leonardo}@cs.utep.edu
and
Shirley Moore
University of Tennessee-Knoxville
shirley@cs.utk.edu

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 event 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 accuracy issues raised by each. Implications for the PAPI cross-platform hardware counter interface and the application programmer also are 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 information such as data and instruction addresses for an event, and pipeline or memory latencies for an instruction. The performance monitoring registers usually are 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. As shown in Table 1, platforms vary in their hardware and operating system support for the two modes, as well as in the number of events that can be monitored. The counter interfaces on the MIPS R12K and IBM AIX Power3, primarily support counting mode. The Compaq Alpha, primarily supports profiling mode through the DCPI interface. Other platforms such as the IA-64 and the X86, 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 sampling, event counts can be estimated by aggregating sample data. However,

the degree of platform support for a particular mode can greatly affect the accuracy of that mode.

Platform	Number of Counters	Number of Events	Modes
MIPS R12K	2	32	C, S(ssrun)
IBM Power3	8	100+	C, S(AIXtrace)
Linux/IA-64	4	150	C, S
Linux/Pentium	2	80+	C, S

Table 1. Processor Counters, Events, and Monitoring Modes.

The Modes column of the table indicates the modes supported by the processor: C indicates support for direct counting of events and S indicates support for sampling of events.

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 related to 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 information presented in this paper, as well as our results from ongoing research, are meant to facilitate performance-tuning efforts that employ aggregate event counts. Inflated counts due to the PAPI interface or other sources are quantified for some events, while what appear to be errors are quantified for other events. For sampling mode, a method of estimating an event count is presented and its accuracy w.r.t. the frequency of an event and the sampling interval is discussed. One may conjecture that aggregate counts are error-free, and that unexpected results, i.e., results that appear to be errors, are due to misunderstandings of processor functionality. If this is the case for some of the results presented herein, we welcome information from vendors that will elucidate the source of the results. Without an understanding of the idiosyncratic behavior of performance counters, an application programmer using such counts could make invalid assumptions about how to tune performance. The remainder of the paper is organized as follows: Section 2 discusses usage models of hardware performance monitoring. Section 3 discusses accuracy issues. Sections 4 and 5 explore implications for the PAPI interface and for the application programmer. Section 6 gives conclusions and describes plans for future work.

2 Usage Models

There are basically two models for using performance-monitoring hardware:

- the *counting* mode, for obtaining aggregate counts of occurrences of specific events, and
- the *sampling mode*, for determining the frequencies of event occurrences produced by program locations at the function, basic block, and/or instruction levels, as well as other information associated with sampled events.

The first step in performance analysis is to measure the aggregate performance characteristics of the application or system under study [7, 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 misprediction rates, can be calculated by dividing counts by the elapsed time.

Sampling mode, together with determination of program locations for sampled events, 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 counts of any two selected events for the R10K (or R12K) hardware event counters, or to time-multiplex all 32 countable events and report extrapolated totals. These data are useful for identifying what performance problems exist (e.g., poor cache behavior identified by a 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` [16] and `HPCView` [6] 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 example, the methodology described in [15] 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.

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 that 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 is then combined in algebraic equations to produce performance predictions.

3 Accuracy Issues

Previous work has shown that aggregate count data may not be accurate, for example when the granularity of the measured code is insufficient to ensure that the overhead introduced by counter interfaces does not dominate the event counts [8]. The analysis in [8] made use of three micro-benchmarks to study eight MIPS R12K events. Currently we are extending this type of analysis to four platforms: MIPS R12K, IBM Power3, Linux/IA-64, and Linux/x86, and nine events: number of loads, stores, floating-point operations, and instructions executed, number of L1 I-cache, L1 D-cache, L2 cache, and TLB misses, and number of branch mispredictions [9].

3.1 Methodology

The methodology used to study the accuracy of performance counters is similar to that used in [8]. It comprises seven phases, which are repeated as is necessary. For a specific event, the seven phases are as follows:

1. design and implement a validation micro-benchmark that permits event count prediction,
2. predict event count using tools and/or mathematical models,
3. collect (hardware reported) event count data using PAPI,
4. collect event count data using a simulator (not always necessary or possible),
5. compare predicted and reported event counts and, when applicable, simulated event counts,
6. analyze results to identify and possibly quantify differences, and
7. when analysis indicates that prediction is not possible, use an alternate means to either verify reported event count accuracy or demonstrate that the reported event count seems reasonable.

In order to design an effective validation micro-benchmark, knowledge of the microarchitecture's or memory hierarchy's structure and management algorithms is needed, but often this information is undisclosed by the manufacturer. In such cases, first *configuration micro-benchmarks* are designed and developed; these benchmarks are used to gain the insights needed about the microarchitecture and/or memory hierarchy. For example, on some of the platforms studied, a validation micro-benchmark that could be used to predict TLB miss event counts could not be designed until configuration micro-benchmarks were designed and developed to understand when and how a TLB miss event occurs. Validation and configuration micro-benchmarks are discussed in more detail in the following two sections.

3.2 Validation Micro-benchmarks

A *validation micro-benchmark* is a simple program, usually small in size, designed to permit prediction of a particular event count; as a result, it stresses a portion of the microarchitecture or memory hierarchy. The micro-benchmark's size, simplicity, or execution time facilitates the tracing of its execution path and/or prediction of the number of times the event is generated. For the events studied in this paper, four validation micro-benchmarks, and variations thereof, are used: array, loop, in-line, and floating-point. Each validation micro-benchmark has many versions, which differ in terms of the number of times the event-of-interest is generated.

3.2.1 Array Validation Micro-benchmark

In the *array validation micro-benchmark* elements of a large array (the size of which differs for different versions of the benchmark) are accessed to stress the data portion of the memory hierarchy and predict associated events, e.g., L1 and L2 data cache miss events and data TLB miss events. The benchmark, in its simplest form, is depicted in Figure 1. In this basic form of the benchmark each datum is accessed once, in sequence (with a stride of one word). Assuming that a cache miss is counted for each cache line accessed that is not in the L1 data cache and prefetching into the cache is not implemented, one L1 data cache miss would be generated for each cache line accessed during the execution of the benchmark. For a processor with a 32-bit word, a 4-Kbyte cache, and a cache line of 8 words, 128 L1 cache misses would be generated for 1024 data accesses. Since there is no reuse of the data, each miss is compulsory and the replacement policy does not factor into the results. If the execution of the micro-benchmark reused cache lines and resulted in cache-line replacements (by the process executing the micro-benchmark or other processes), then event count prediction would be difficult if not impossible. For example, consider a cache with a random replacement policy. Variations of this micro-benchmark access the array with different size strides and with a random access pattern.

```
for( i = 0; i < array_size; i++)
{
    a[i] = 1;
}
```

Figure 1. Array Validation Micro-benchmark.

3.2.2 Loop Validation Micro-benchmark

The *loop validation micro-benchmark*, shown in Figure 2 in its simplest form, consists of a sequence of instructions within a loop. With such a benchmark, it is easy to stress a particular functional unit, e.g., to count the number of stores executed. The instruction sequence is constructed so that data dependencies among the instructions ensure that the compiler does not reorder nor optimize the sequence of instructions. This benchmark was used to count the number of loads, stores, and instructions executed.

For the loop validation micro-benchmarks referred to in this paper, the depicted pattern was repeated to construct a loop body of 100 instructions and different versions of the benchmark were created by increasing the number of loop iterations executed. For this benchmark and the targeted events, the source count method, described in Section 3.4, was used to predict event counts. First, the code was compiled to an assembler program. Then, using a Unix script, the number of load instructions, store instructions, and total instructions in the file were counted. Finally, these counts were multiplied by the number of loop iterations to compute the event count predictions.

```

for (i=0; i<number_of_loops; i++)
{
    a = 1;
    b = 1;
    c = 1;
    a = b + 1;
    b = a + 1;
    c = a + b;
    a = b + c;
    b = a + c;
    c = a + b;
    a = 1;
    ...
}

```

Figure 2. Loop Validation Micro-benchmark.

3.2.3 In-line Validation Micro-benchmark

The *in-line validation micro-benchmark* is the unrolled version of the loop benchmark. This benchmark is designed to stress the instruction memory hierarchy and predict the number of instruction cache misses, as well as, the number of instructions executed. Versions of this benchmark with code sizes of 100 to 1,000,000 instructions were used. For event count predictions the code was compiled to the assembler level and the number of instructions counted.

3.2.4 Floating-Point Validation Micro-benchmark

The *floating-point validation micro-benchmark*, depicted in Figure 3, has the same basic form as the loop validation micro-benchmark, however, the integer values are replaced by floating-point values. Optimization on some platforms required that the constant 1.0 be replaced by an input parameter *FP_val*. The floating-point and loop validation micro-benchmarks used to obtain the results reported in this paper have the same design characteristics; in addition, the method used to compute event count predictions is the same in both cases.

```

for (i=0; i<number_of_loops; i++)
{
    a = FP_val;
    b = FP_val;
    c = FP_val;
    a = b + FP_val;
    b = a + FP_val;
    c = a + b;
    a = b + c;
    b = a + c;
    c = a + b;
    a = FP_val;
}

```

Figure 3. Floating-point Validation Micro-benchmark.

3.3 Configuration Micro-benchmarks

A *configuration micro-benchmark* is a program designed to provide insight into the structure and management algorithms of the microarchitecture and/or memory hierarchy. Its purpose is to provide the information required to design a validation micro-benchmark or to refine its characteristic predictability. A configuration micro-benchmark is not always necessary or appropriate for all platforms/events under study; however, these benchmarks are indispensable in creating effective validation micro-benchmarks when limited documentation is available.

An example of a configuration micro-benchmark is one that was needed to design an array validation micro-benchmark that allows prediction of data TLB miss events. This benchmark consists of code that executes one traversal of an array of integers, accessing the array at regular strides. The array resides in multiple pages so that the traversal generates a predictable number of compulsory data TLB misses. In order to design this benchmark two pieces of information were needed: 1) size of the pages that store the array (some platforms, such as the R10K, the Itanium and the Power3, support multiple page sizes) and 2) identification of the location of the first byte of a page frame with respect to the beginning of the array data structure. The page size and the stride with which the array is accessed determine the rate at which the validation micro-benchmark generates TLB misses. The identification of the first byte of a page frame is needed to perfect the prediction of total TLB misses generated by the validation micro-benchmark.

To identify the page size used to store the array and verify TLB size, a configuration micro-benchmark was implemented using Saavedra's methodology [14]. This micro-benchmark accesses an array using a stride that increases at each iteration and records average read/write reference times. Several memory hierarchy characteristics, including data cache line size and associativity, page size, data TLB associativity and reachability, as well as corresponding latencies, can be deduced from graphing the average reference times.

To identify the starting address of the array w.r.t. the beginning of a page frame, a second configuration micro-benchmark was designed and implemented. This information was needed since, ideally, the validation micro-benchmark makes array references that align exactly to the beginning of a page frame or at some known offset from it. This configuration benchmark, called the "padding benchmark", provides the information needed to identify the offset from the starting address of the array that forces this alignment in the validation benchmark. The configuration benchmark traverses the array, referencing the first and last array elements on every-other page-sized region; the page-sized regions are defined by an offset or "padding" from the starting address of the array. Via PAPI, the number of generated data TLB misses is recorded; successive variations of the benchmark use larger offsets. The data TLB miss counts reported for each variation of the benchmark identify the offset that will force aligned references in the validation benchmark, i.e., the offset that produces a single TLB miss for each pair of references. If pairs of references are not aligned, each pair will necessarily map to consecutive page frames and produce two misses. Figure 4 shows the results, for three different platforms for which page sizes are known, of executing the "padding benchmark" for 100 pairs of

references that traverse the array. On the R10K, which uses 32-KByte pages, the recorded TLB miss count for all offsets except 28012 bytes was approximately 200; for an offset of 28012 bytes, the count dropped to approximately half this amount. On the Itanium, which uses 16 K-Byte pages, the recorded TLB miss count for all offsets except 16364 bytes was approximately 1000, dropping to half this amount for the offset of 16364. Finally, on the Power3, which uses 4 K-Byte pages, the recorded TLB miss count was 600 for all offsets except 2260 bytes, at which point it dropped by half this amount.

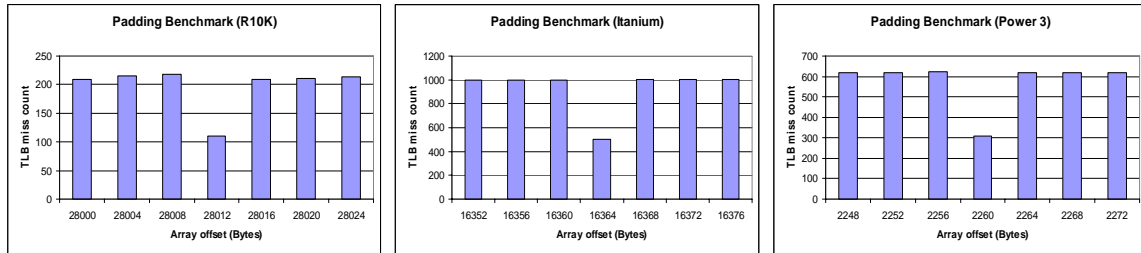


Figure 4. Padding Benchmark Results for 100 Iterations.

Figure 5 presents, for the three platforms, a comparison of the data TLB miss event counts predicted by the validation micro-benchmarks (designed using the information identified by the configuration benchmarks described above) and the event counts reported by the hardware; the y-axis represents the percentage difference between the reported count and the predicted count, and the x-axis represents the number of array references. As shown, for the R10K platform, the difference between predicted and reported counts ranges between 4% and 14%, the predicted being lower than the reported and the difference being less significant for larger numbers of array references. At least part of the difference is likely attributable to PAPI. For the Itanium platform, the difference is consistently around 400%, which represents a consistent difference that is five times the predicted count. This type of “multiplicative” difference is discussed in Section 3.6.1.2. On the Power3, multiplicative differences also are observed; for small counts the difference is near 600%, while for larger counts it is approximately 200%, which indicates a multiplicative difference of three times the predicted count.

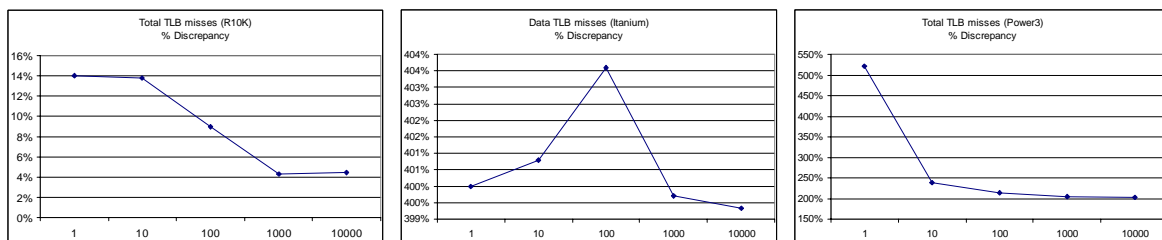


Figure 5. TLB Miss Validation Benchmark Results.

3.4 Event Count Prediction

To determine event count accuracy, a reliable event count prediction, i.e., an expected (event count) value, for the associated micro-benchmark is needed. The expected value is derived using one or more methods including: 1) direct counting, 2) mathematical modeling, and 3) simulation results. The particular method used depends on the benchmark and the platform being studied. The *direct count method* employs a script or program that parses the assembler code and counts the event of interest, e.g., loads, stores, or total instructions executed.

Mathematical models are used when events take place at regular intervals, e.g., cache and TLB misses. In some cases, using the size and organization of the cache and the data access pattern of the benchmark, the expected number of cache misses can be calculated. In other cases the microarchitecture of the processor must be considered to arrive at a suitable mathematical model. For instance, the Itanium instruction is 41 bits long; three instructions are fetched at once and a 5-bit header is added to the ‘bundle’ for a total of 128 bits. Some instructions are not bundled with other instructions (or not bundled in the order required by the program) and the bundle may be padded with *nop* instructions. The number of bundles that the processor is required to fetch during the execution of the benchmark must be calculated in order to predict the number of instruction cache lines used and hence the number of instruction cache misses. As discussed below, using mathematical modeling in this way may not suffice when sophisticated, often proprietary, algorithms are implemented by the manufacturers to ameliorate memory hierarchy latency. In these cases, as demonstrated below, mathematical modeling can be used to indicate whether or not the events counts seem reasonable, e.g, execution time grows proportionately with the event count.

Simulators can be used to verify memory hierarchy events. This method was demonstrated in [8] and is described below w.r.t. L1 data cache miss event prediction.

The feasibility of event count prediction and the ease with which a prediction can be made differs across the spectrum of events. For example, events that count the number of times a particular instruction type, such as a load or floating-point instruction, is executed can be predicted easily. In this case, a tool was designed and developed to count the number of occurrences of the instruction type in the assembler code. In contrast, counts of memory hierarchy events, such as cache and TLB misses, can be difficult and may be impossible to predict given the sophisticated algorithms used by manufacturers to optimize memory performance. For example, if sufficient information is not known about the configuration and implementation of the stream buffers used to prefetch data, it may be possible to reliably predict data cache miss event counts only when the benchmark accesses data in a random pattern. Similarly, when a random TLB replacement policy is implemented, it may only be possible to predict TLB miss event counts when replacements do not occur. Alternatively, event counts can be shown to be reliable by observing a relationship between the growth of execution time and the growth of the event count. However, in cases such as these, it is not clear how helpful the performance counter information is for the application programmer, i.e., how useful it is w.r.t. tuning the performance of his/her application.

As an example, consider L1 data cache misses. Since three of the four processor architectures studied would seem to implement a sophisticated data prefetch algorithm,

which is not in the public domain (to our knowledge), it is not possible to reliably predict the L1 data cache miss event count unless the prefetch algorithm is foiled. In the absence of a micro-benchmark that does this, given that some events, such as cache misses, are relatively expensive in terms of time, execution time can be measured to gain a sense of whether or not the reported event counts are reasonable.

Experiments indicated that the prefetching algorithm is quite sophisticated. Using the array micro-benchmark (which accesses the array in strides of one word), predictions that either do not consider prefetching or consider a simple prefetch algorithm, such as fetch two cache lines on a miss, were much higher than the reported event counts. In fact, on some platforms, as the size of the array and the number of accesses was increased, the miss rate approached 0. Using another variation of the array benchmark that accesses the array in strides of size two cache-size-units plus one cache-line, the reported event counts were nearly identical to those for the original array benchmark. (This variation of the benchmark was run under the mistaken assumption that it would foil the prefetching algorithms, the idea being that while the references occur at regular intervals, the reference pattern is not computed by simple adds or multiplies. This proved to be a naive approach.)

Since the prefetching algorithm was difficult to identify, another version of the array microbenchmark was employed; this version attempts to foil the prefetch scheme. This benchmark contains a random number generator that is used to generate a random access pattern. The access pattern is captured for later use in a cache simulator to derive the predicted number of cache misses. The benchmark follows:

```
for( i = 0; i < array_size; i++)
{
    r = random();
    scale r to array_size;
    a[r] = 1;
}
```

Figure 6. Random-access Array Micro-benchmark.

To predict event counts, a cache simulator was developed and initialized to model the size and configuration of the caches of the platforms being studied. Assuming that prefetching had been foiled, there was no need simulate a prefetch algorithm. A least-recently used policy was used to determine what line to replace in the n-way set associative caches studied. Using this methodology, for all studied platforms except the Itanium and for array sizes larger than one cache-size-unit, the predicted L1 data cache miss counts were within 10% of the reported event counts. For the Itanium this level of predictability did not occur until an array size of about 10 times the cache size. For array sizes smaller than one cache-size-unit the number of reported misses starts out greater than the predicted but quickly drops to less than the predicted for array sizes larger than about 25% of the cache size.

In this case it also is possible to understand whether or not the reported event counts seem reasonable. This was done by verifying that the cycle count (execution time) increases proportionately with the increase in cache misses. The following formula quantifies the cycles fairly consistently:

$$\text{total_number_of_cycles} = \text{iterations} * \text{exec_cycles_per_iteration} + \text{cache_misses} * \text{cycles_per_cache_miss}$$

where *cycles_per_cache_miss* is an average measure that includes all memory hierarchy miss time.

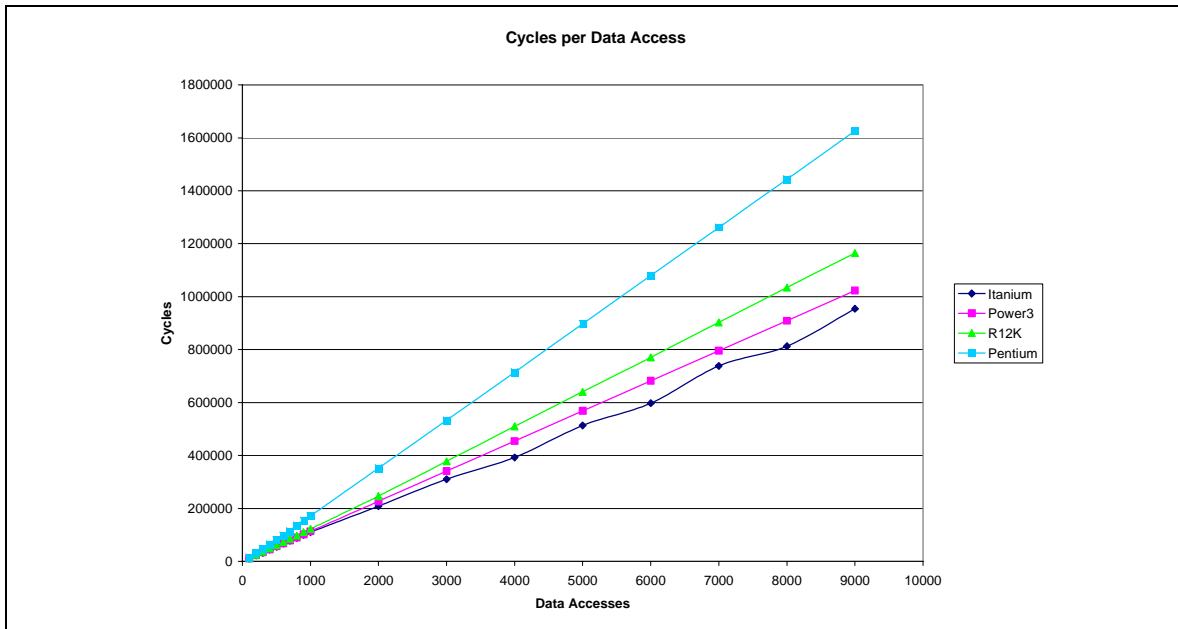


Figure 7. Number of Cycles Increasing with Number of Data Access/Cache Misses.

To use this formula, the average number of execution cycles per iteration and the average number of cycles per cache miss must be known. Executing the benchmark, using PAPI, the total number of cycles and the number of cache misses can be accessed. In this case, for one platform, using two versions of the array micro-benchmark, one that accesses 1000 data elements and another that accesses 2000, it was found that the number of cache misses was identical and, of course, the total number of cycles was different. Thus, the difference in the total number of cycles was due only to execution of the code, while the remainder was due to other events. In this case, since the code size is small (the number of instruction cache misses is small) and the number of reported TLB misses is very small, the majority of the “other events” are associated with L1 and L2 data cache misses. Accordingly, *exec_cycles_per_iteration* and *cycles_per_cache_miss* were computed. As shown in Figure 7, using this formula, it was determined that the total number of cycles does, indeed, increase proportionately with the increase in cache misses (in this case, data

accesses). In general, the predicted execution time was within 5% of the reported execution time.

3.5 Reported Event Count Collection

As previously described, micro-benchmarks are designed to facilitate execution path tracing and event count prediction. These benchmarks also serve another purpose when it comes to reported event count collection. A small benchmark increases the probability that code execution will be limited to a single time slice and, therefore, limits the perturbation that might be introduced by the operating system. This is important because even though in some cases events generated on behalf of the process under study and other processes are segregated, there are some events that may be severely affected by the processing environment. For example, if the execution of the benchmark code spans multiple time slices, then event counts associated with the cache hierarchy may be perturbed by the cache activity of other processes. Furthermore, since the micro-benchmarks are designed to facilitate the modification of the number of events that will be generated, it is easy to generate a significant event count such that renders overhead insignificant.

When collecting reported event counts, in order to limit the potential perturbation of other processes and to verify the overhead associated with the PAPI interface, an instrumented benchmark is run 100 times via a script. Then the mean event count is calculated and used as event count to compare to the predicted event count. To determine the stability of the collected data, the standard deviation is computed. The standard deviation identifies possible anomalies (for an example, see the section that follows) and triggers further data analysis.

3.6 Data Analysis

Different accuracy issues are associated with counting and sampling modes. Section 3.6.1 addresses those associated with counting mode, i.e., differences between reported and predicted event counts. Section 3.6.2 focuses on accuracy issues associated with sampling mode, in particular, statistical errors.

3.6.1 Aggregate Counts

Using the methodology discussed in Section 3.1, four types of differences between reported and predicted aggregate event counts were identified: overhead or bias, multiplicative, random, and unknown.

3.6.1.1 Overhead or Bias

The first type of difference, *overhead or bias*, denotes a constant difference between the predicted and reported counts that is attributable to the interface, in this case, PAPI, and/or some other unidentified source. The interface sets up and starts the performance counters before the execution of the targeted code and stops the counters and reads the reported event count after code execution. The overhead associated with the event counts for the number of loads and stores executed on the processors of interest are identified in Table 2. If the event count is large, the overhead will not be significant. However, calibration can be achieved even if the event count is small and thus, an accurate count attained, by subtracting the overhead from the reported count. In this case,

another way to attain an accurate event count is to repeat the execution of the targeted code many times, without starting and stopping the counter, so that the expected value is very large compared to the overhead and thus, the difference is insignificant.

Platform	MIPS R12K	IBM Power3	Linux/IA-64	Linux/Pentium
Loads	46	28	86	N/A
Stores	Multiplicative Difference	31	129	N/A

Table 2. Overhead Differences Associated with Load and Store Events.

3.6.1.2. Multiplicative Differences

The second type of difference, *multiplicative*, is associated with a reported event count that exceeds the predicted count by a defined factor. For example, for the R12K, IA-64, and Pentium the reported event counts for the number of floating-point operations in a floating-point validation benchmark containing only floating-point additions are equal to the predicted counts. In contrast, the reported event count is twice that of the predicted count for the Power3. In this case, the larger the event count, the larger the difference; however, calibration can be achieved easily by dividing by the factor, in this case two. Note that an additional source of difference, i.e. overhead, may be incorporated in what appears to be a multiplicative difference, e.g., $reported_count = factor * predicted_count + overhead$. In such cases the reported count can still be useful if the factor and overhead are known.

3.6.1.3 Random Differences

The third type of difference, *random*, occurs when the reported event count differs significantly from the predicted count, but only part of the time. For example, for the IA-64, L1 D-cache differences of three orders of magnitude were observed about 8% of the time. Such a random difference can make the event count unusable. If the targeted code is run repeatedly, and the event counts averaged, such large differences will skew the average to the point of unreliability. Removing the ‘outliers’ will produce a reported event count that is more in line with the predicted count. However, these outliers are cause for concern since the use of a performance count with such a large difference can be totally useless, or worse, detrimental to performance tuning.

3.6.1.4 Unknown Differences

The fourth type of difference, *unknown*, occurs when we simply do not know how the processor is behaving or the difference is attributable to multiple sources. One example occurs when counting L1 data cache misses.

As mentioned in Section 3.4, all of the processors of interest in this paper use some type of data prefetch mechanism. The mechanisms may include hardware such as one or more stream buffers as well as algorithms that govern the behavior of the hardware. While some manufacturers publish partial information about the hardware used, they are universally reluctant to release details of the algorithms used. Without knowledge of how the processor functions, it is very difficult, if not practically impossible, to predict event

counts associated with cache misses when prefetching is effective. Nonetheless, other methods may be employed to determine if the results are reasonable. For example (as was described in Section 3.4), in this case, the array validation micro-benchmark was used to show that the cycle count increased proportionately to the increase in cache misses and a variation of this micro-benchmark was designed that foiled the prefetching scheme, permitting fairly accurate prediction of event counts. Of course, overhead is hidden in these counts.

In contrast to the differences associated with L1 data cache miss event counts, there are differences associated with other events that are neither predictable nor verifiable. One example is the event count associated with branch mispredictions.

3.6.2 Sampling Errors

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 [4, 5]. An instruction is selected 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 [7]. 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 that 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 re-implemented. 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 that 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 execute 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 Implications for the Application Programmer

The only studied event count for which reported and predicted counts are equal is that associated with the total number of floating-point operations; this is true for three of the four platforms and is due, in part, to the fact that the PAPI interface code contains no floating-point operations. As described above, for the Power3 it is calibrated easily. This level of accuracy makes these counts very useable for performance tuning. (Note that only floating-point add operations have been studied thus far; experiments using the various types of floating-point operations are in progress as well as experiments with a mix of these operations.)

The next most useable of the studied event counts are those that count the number of executed loads and stores. As shown in Figure 8, for two platforms these counts had definable overheads and for all three platforms studied they were within 1% of predicted counts for all but the smallest of the versions of the loop validation micro-benchmarks.

Similarly, as shown in Figure 9, both the Pentium and Power3 have definable overheads for the event counts associated with total number of instructions retired. For these platforms as well as the R12K reported event counts were within 1% of predicted counts. In contrast, the Itanium reported counts were consistently about 17% larger than predicted counts (see Figure 9).

Although cache miss counts are an important metric for performance tuning, it is not possible to attain an accurate count via calibration. For example, the overhead associated with the PAPI interface code cannot be defined since its degree of perturbation depends on the nature of the code being monitored. Results of using the in-line micro-benchmark show that the usefulness of L1 instruction cache miss event counts differs from platform to platform. For the R12K, reported counts are within 1% of predicted counts, while the

Itanium reported counts are about 17% greater than the predicted counts. In contrast, the Power3 reported counts are ~55% lower than the predicted counts. From these results, it appears that L1 instruction cache event counts on the Power3 are useful only with more information from the vendor about the instruction prefetch algorithm employed.

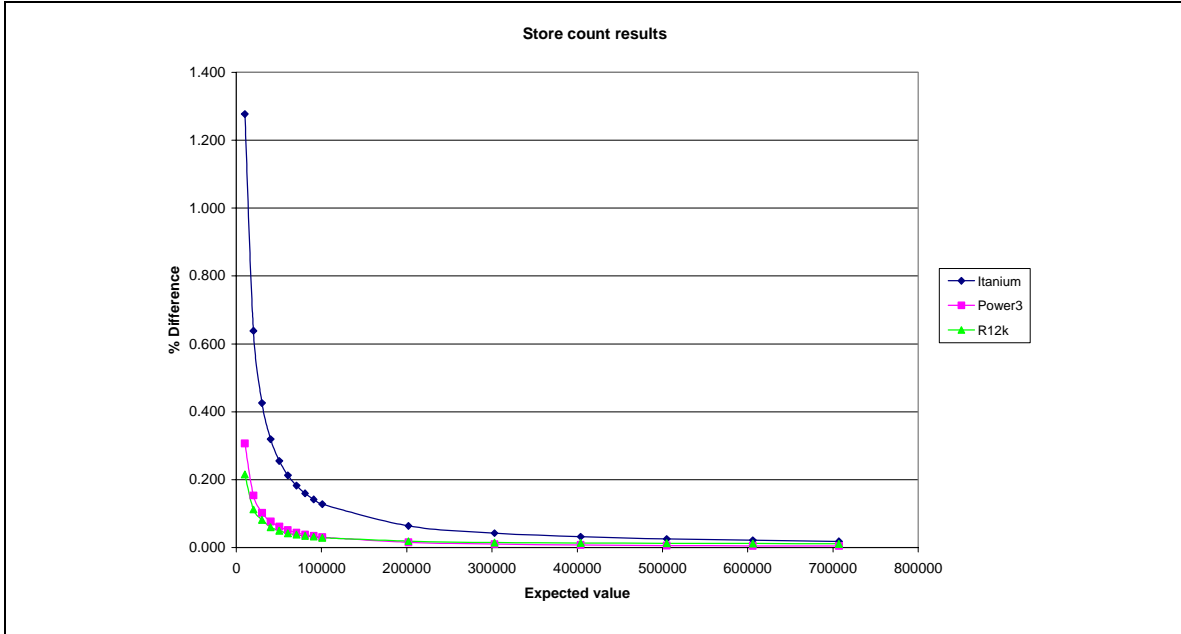


Figure 8. Stores Executed.

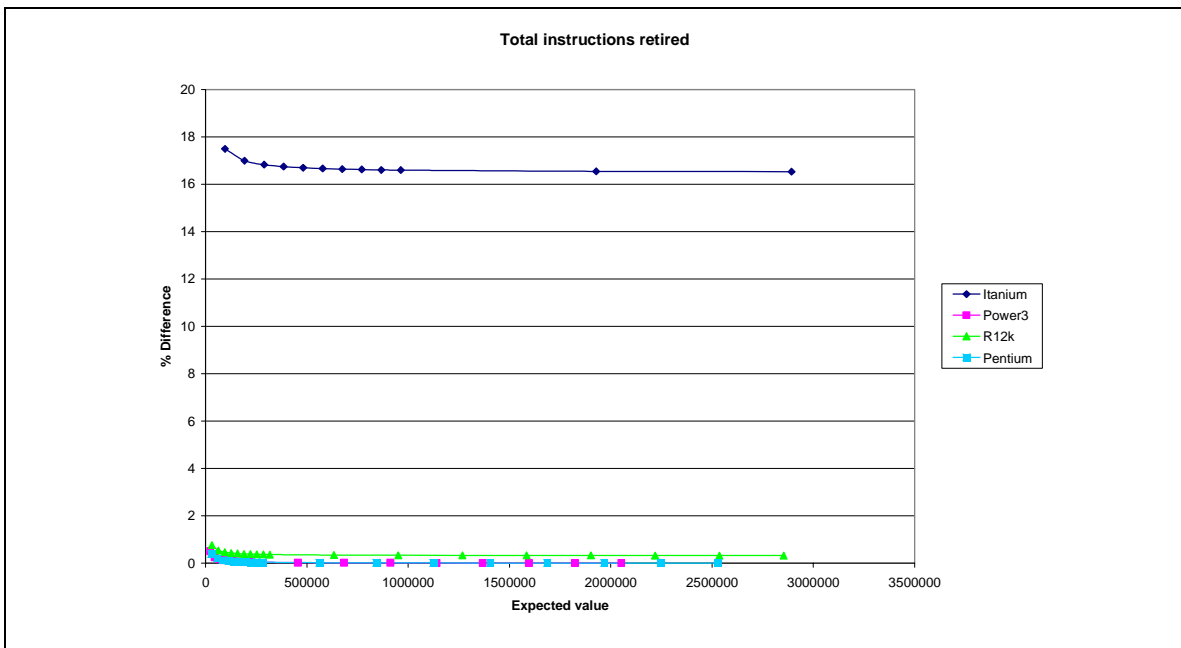


Figure 9. Instructions Retired.

As described in Section 3.4, due to lack of knowledge about the implemented data prefetch schemes, the L1 data cache miss event counts proved to be unpredictable in normal use, i.e., when not randomly accessing data (see Figure 10). However, they appear to be reasonable based on our verification method.

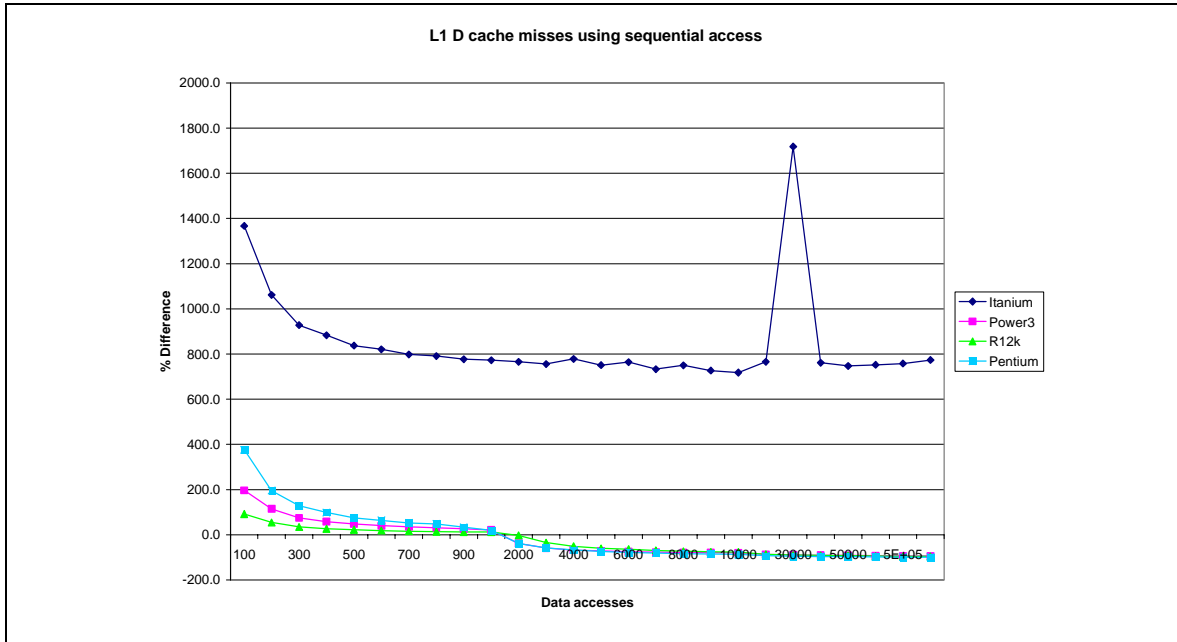


Figure 10. L1 Data Cache Misses - Sequential Access.

6 Conclusions and Future Work

It is clear that both counting and sampling modes of 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_preadread`) 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 micro-benchmarks as in [8], 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 sampling data on different platforms, and to investigate the feasibility and accuracy of implementing counting mode in software on top of hardware-supported sampling 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., J. Dongarra, N. Garner, G. Ho, and P. Mucci. "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, 14:3, Fall 2000, pp. 189-204.
- [2] Browne, S., J. Dongarra, N. Garner, K. London, and P. Mucci. "A Scalable Cross-Platform Infrastructure for Application Performance Optimization Using Hardware Counters," *Proceedings of SC'2000*, Dallas, TX, November 2000.
- [3] Buck, B., and J. K. Hollingsworth. "Using Hardware Performance Monitors to Isolate Memory Bottlenecks," *Proceedings of SC'2000*, Dallas, TX, November 2000.
- [4] Dean, J., J. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proceedings of the 30th Symposium on Microarchitecture (Micro-30)*, December 1997.
- [5] Dean, J., C. A. Waldspurger, and W. E. Weihl. "Transparent, Low-Overhead Profiling on Modern Processors," *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998.
- [6] HPCView: <http://www.cs.rice.edu/~{ }dssystem/hpcview/>
- [7] *Intel IA-64 Architecture Software Developer's Manual, Volume 4: Itanium Processor Programmer's Guide*. Intel, July 2000. <http://developer.intel.com/>
- [8] Korn, W., P. Teller, G. Castillo. "Just How Accurate Are Performance Counters?," *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, April 2001.
- [9] Maxwell, M. "Understanding Microprocessor Performance Event Counts," Master's Thesis, Department of Computer Science, The University of Texas at El Paso, in progress (expected completion date: December 2002).
- [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," *Proceedings of the 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] Saavedra, R., and A. J. Smith "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes," *IEEE Transactions on Computers*, 44:10, October 1995.
- [15] Snively, A., N. Wolter, and L. Carrington. "Modeling Application Performance by Convolving Machine Signatures with Application Profiles," *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.
- [16] The Visual Profiler: <http://aros.ca.sandia.gov/~{ }cljanss/perf/vprof/>