Towards a Cross-Platform Microbenchmark Suite for Evaluating Hardware Performance Counter Data

Maria Gabriela Aguilera, Roberto Araiza, Thientam Pham, and Patricia J. Teller
Department of Computer Science
University of Texas-El Paso
500 W. University Drive
El Paso, TX 79968 USA
(915)747-8012
{maguilera, raraiza, tpham, pteller}@cs.utep.edu

Keywords – Performance monitoring hardware counters, benchmarking, performance validation.

Abstract

As useful as performance counters are, the meaning of reported aggregate event counts is sometimes questionable. Questions arise due to unanticipated processor behavior, overhead associated with the interface, the granularity of the monitored code, hardware errors, and lack of standards with respect to event definitions. To explore these issues, we are conducting a sequence of studies using carefully crafted microbenchmarks that permit the accurate prediction of event counts and investigation of the differences between hardware-reported and predicted event counts. This paper presents the methodology employed, some of the microbenchmarks developed, and some of the information uncovered to date. The information provided by this work allows application developers to better understand the data provided by hardware performance counters and better utilize it to tune application performance. A goal of this research is to develop a cross-platform microbenchmark suite that can be used by application developers for these purposes. Some of the microbenchmarks in this suite are discussed in the paper.

1. Introduction

Performance monitoring hardware consists of a set of registers that record information about different processor events that occur during application execution. This information is in the form of aggregate event counts or sampled event traces. For example, when used in the former manner, the registers accumulate counts of the occurrences of events triggered by memory hierarchy activity, such as level-one data-cache misses or translation-lookaside buffer misses, or functional unit activity, such as the execution of floating-point or branch instructions. When used to produce sampled event traces, the registers collect, at a defined frequency, detailed information about the occurrence of the specified event, for example, processor id, thread id, timestamp, effective instruction address, and data address. Processors with hardware performance counters include the DEC Alpha, IBM Power, Intel Pentium, and Sun UltraSPARC series [7]. As shown in Table 1, the number of counters available for performance monitoring, the number and types of events that can be monitored, and the number of events that can be monitored concurrently vary among the different platforms. Note that the starred platforms are the ones that are the targets of our study, i.e., the platforms of interest.

Different manufacturers provide software that interfaces with hardware performance counters on their processors. Higher-level user interfaces, such as the Performance Application Programming Interface (PAPI) [1], the Hardware Performance Monitor (HPM) Tool Kit [2], and the Hardware Activity Reporter (HAR) [14] facilitate access to performance counters. For example, PAPI provides a cross-platform user interface to access performance counters on various processors. It can be used to monitor a set of 104 different events. Note, however, that no platform supports all 104 PAPI events. For example, the Pentium II and Pentium III processors support 49 of these events, while the Power4 supports 22.

Hardware performance counters are used to analyze application performance [6, 8]. For instance, via an end-user tool that provides a graphical view of performance information [8], called perfometer, hardware performance counters are used to identify application performance bottlenecks. Performance counters also are used to examine memory system behavior in the context of the target application's data space [6]. Furthermore, performance counters are used to monitor security application behavior and

identify odd usage patterns that can be used to detect security breaches as they occur. By doing this, events such as Distributed Denial of Service (DDOS) attacks can be identified [12].

As useful as performance counters are, the meaning of reported aggregate event counts is sometimes questionable. Questions arise due to unanticipated processor behavior, overhead associated with the interface, the granularity of the monitored code, hardware errors, and lack of standards w.r.t. event definitions. To explore these issues, we are conducting a sequence of studies using carefully-crafted microbenchmarks that permit the accurate prediction of event counts and investigation of the differences between hardware-reported and predicted event counts. This paper presents the methodology employed, some of the microbenchmarks developed, and some of the information uncovered to date. The information provided by this work allows application developers to better understand the data provided by hardware performance counters and better utilize it to tune application performance. A goal of this research is to develop a cross-platform microbenchmark suite that can be used by application developers for these purposes. Some of the microbenchmarks in this suite are discussed in the paper.

Processor	Number of Counters	Number of Native Events
AMD Athlon	4	24
DEC Alpha 21x64	2	44
Intel Itanium *	4	153
Intel Pentium III*	2	78
Intel Pentium IV	18	39
IBM Power3-II*	8	198
IBM Power4*	8	244
SGI MIPS R12K*	32	32
SGI MIPS R14K	2	32
Sun UltraSparc II	2	22

Table 1. Performance counters on various processors

After describing our experimental and microbenchmark design methodologies in the next two sections, Section 4 concentrates on events studied. In this latter section, each event is the focus of a subsection, which defines the event under study, the challenges involved in designing a microbenchmark that can be used across platforms, the microbenchmark used to study the event, and the results of the

study across a number of platforms. The last section of the paper presents the status of this work, conclusions, and future work.

2. Experimental Methodology – Predicted vs. Reported Event Counts

The methodology used to study aggregate event counts, which was presented in [7, 9, 15], has been refined. Now it prescribes microbenchmarks that are parameterized and have built-in scalability, and it includes a feedback loop. In this context, parameterization and scalability are meant to ease experimentation across platforms and a range of increasingly larger event counts. These modifications to the methodology enhance microbenchmark design and implementation.

The methodology, depicted in Figure 1, applied to a particular event, includes the following steps:

1. design and implement, when possible, a parameterized, scalable microbenchmark that permits event count predictions;
2. predict event counts for a range of increasingly larger event counts using tools and mathematical models developed with knowledge of the architecture, operating system, and compiler;
3. collect hardware-reported event counts using PAPI or another API;
4. compare predicted and reported event counts; and
5. analyze experimental results to identify and quantify differences between predicted and reported counts; if necessary repeat all five steps to reflect the knowledge gained from the analysis.

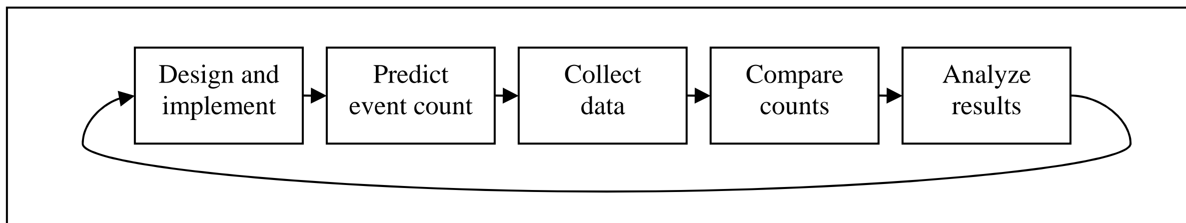

Figure 1. Experimental methodology

Various techniques are used in each step. For instance, when designing and implementing microbenchmarks that measure cache or TLB misses, it is necessary to identify the configuration of the memory hierarchy of the targeted platform. This can be done in three different ways:

1. Consult processor documentation: This approach may be thwarted by documentation being unavailable, or being too generic to get the necessary details.
2. Use a program that queries the processor for this information: An example of this is. *mem_info.c*, which is included with the PAPI software.
3. Design and implement a configuration microbenchmark, which, through memory access behavior, deduces the memory hierarchy configuration [15].

Microbenchmark Design and Event Count Prediction: To predict and collect event counts, a variety of microbenchmark designs are used. Some microbenchmarks, like those that measure cache misses, are designed to produce the desired number of misses via an input parameter. Others, like those that target floating-point load or store instructions, require counting a class of either executed assembler or machine instructions. The latter can be counted by examining the disassembled object module, which can be generated using a tool such as *objdump* [4]. When counting by hand is infeasible due to the size of the monitored code, Perl scripts are utilized.

Event Count Collection: Microbenchmarks are written in C, and PAPI is used to collect hardware-reported event counts. A microbenchmark is used to count 1, 10, 100, 1,000, 10,000, 100,000, and 1,000,000 occurrences of an event. For each target event count the microbenchmark is run 100 times to compute and analyze the mean, standard deviation, and variance among different microbenchmark runs and capture anomalies in counter behavior. As shown in Figure 2, PAPI directives, which start and stop the counters, are placed around the C code to be monitored. In general, multiple events can be counted concurrently. This is helpful when counting related events, such as instructions issued and instructions completed, or when there is an unexpected event count that might be a consequence of or may be explained by other event counts. For example, during the design of the cycles microbenchmark, which is discussed in Section 4.3, cache misses were counted to see if they were responsible for unexpected cycle counts.

```
PAPI_start(EventSet);
for (i=1; i<n; i++)
{
        a = a + init_val;
}
PAPI_stop(EventSet, values[0]);
```

Figure 2. PAPI directives that delineate monitored code

Comparison and Analysis of Predicted and Reported Event Counts: Once reported event counts are collected, they are compared to predicted counts. This comparison is done by computing, for each event count granularity, the percent differences between predicted and reported counts. If the predicted and reported counts are not comparable, if appropriate, an attempt is made to quantify interface overhead. If the percent difference is significant and does not appear to be totally attributable to interface overhead, then the results are analyzed to identify the cause of the differences and how the microbenchmark should be modified; then the methodology is repeated. Analysis of results may include the use of low-level interfaces, like the HPM Toolkit [2] and perfex [10], to verify PAPI event counts; validating event definition, i.e., validating if what is counted is what you think is being counted; and employing other event counts to identify sources of differences.

3. Cross-platform Microbenchmark Design Methodology

As mentioned above, microbenchmarks are written in C and use PAPI to collect hardware-reported event counts, and are designed to allow validation of hardware event definitions and prediction of the number of event occurrences. These objectives are meant to ensure that hardware performance counters count what application programmers expect them to count and to explain why sometimes the reported counts are not what are expected. In this context, a microbenchmark is designed according to the following criteria:

1. functionality, in terms of stressing that part of the microarchitecture or memory hierarchy that triggers the target event,
2. compactness, in terms of static size,
3. efficiency, in terms of execution time,

4. simplicity, in terms of the amount of potential concurrency that can be exploited by the microarchitecture and memory subsystem, and
5. portability among microprocessors.

Functionality: A microbenchmark is designed to stress the part of the microarchitecture or memory hierarchy that triggers the target event, e.g., an L1 data-cache miss microbenchmark stresses the load/store functional units and the L1 data cache, and a floating-point instruction microbenchmark stresses the floating-point functional units. Since a microbenchmark is designed based on an understanding of the trigger of an event, unexpected event counts immediately suggest a lack of understanding of the event definition. For example, on the Power3 architecture, unexpected event counts for L1 instruction-cache hits were the first indication that the definition of this event was different from the generally accepted one [13].

Compactness: The smaller the microbenchmark, in terms of its static size, the easier it is to predict target event counts, i.e., the easier it is to analyze the monitored code and estimate event counts.

Efficiency: The smaller the microbenchmark, in terms of its dynamic size, the more likely it will execute within one time quantum and, accordingly, the more likely reported counts will be comparable to predicted counts. This is because for some events, e.g., cache or TLB misses, context switches may introduce measurement perturbation, i.e., intervening processes may evict parts of the monitored process' working set, causing "additional" misses and generally non-repeatable (unpredictable) miss behavior. For other events, like instruction counts, context switches do not introduce perturbations because PAPI operating-system patches cause the state of hardware counters to be saved and restored on context switches of the monitored process.

Simplicity: The simpler the microbenchmark, the less the microarchitecture can exploit parallelism during the execution of the monitored code and, thus, the easier it is to predict event counts. Effects of microarchitecture optimizations, e.g., effects of event concurrency and latency-hiding techniques, make it difficult to predict event counts. For example, as is discussed in Section 4.3, eliminating the generation of

instruction and data cache misses, the use of branch prediction and speculative execution, and the introduction of pipeline stalls, facilitates cycle count prediction.

Portability: Microbenchmark portability comes in two flavors: (1) portability of the microbenchmark source and (2) portability of the microbenchmark design. In the former case, the best case, portability may be achieved directly or indirectly. In the direct case, the microbenchmark is simply ported to the target platform. In the indirect case, the microbenchmark is designed with input parameters, the definition of which are machine specific. Now, porting also requires definition of input parameter values. In each case, the source code remains intact. When portability is restricted to the design of the microbenchmark, the design remains intact across platforms but the implementation, being machine dependent, is modified accordingly. Either the source code or methodology changes with the target processor.

To achieve direct portability of the microbenchmark source, we strive to identify similarities across processor platforms and capitalize on these similarities. For example, even though branch prediction is implemented differently on different platforms, for-loop branches are predicted similarly on all studied platforms. This fact is exploited to develop the high-level cross-platform branch misprediction microbenchmark, which is the subject of Section 4.1.

When direct portability of the microbenchmark source is not possible, indirect portability of the source code may be possible via parameterization. As discussed above, parameterization also is used to permit one microbenchmark to be used to generate different numbers of events.

When portability of the microbenchmark source is not possible, we strive for portability of the microbenchmark design. For example, given the standard definition of an instruction cache miss event, the underlying requirements and specification of the instruction cache miss microbenchmark are applicable to all platforms. However, the implementation depends on processor-dependent design factors, i.e., instruction size and cache configuration. Thus, as shown in Section 4.2, the high-level language microbenchmark must be modified according to these design points. Similarly, as discussed in Section 4.3, the underlying requirements and specification of the microbenchmark used to study the total cycles event is common across platforms, but because of differences in the ISAs (instruction set architectures)

w.r.t. register specification, the methodology used to create the low-level language microbenchmark must be modified.

In summary, when designing a microbenchmark, the similarities across platforms are exploited when possible. In the best case, the high-level language microbenchmark is simply ported to the various platforms. In the worst case, the design is portable but the implementation of the design changes with the platform.

4. Some Microbenchmarks and Results

This section presents examples of the microbenchmarks that we have used to evaluate performance counter data as well as results of these evaluation studies. Each subsection focuses on a microbenchmark, defining the event under study, the event count prediction method, the challenges involved in designing the microbenchmark, the microbenchmark that was used to study the event, and the results of the study across a number of platforms.

4.1 Branch Misprediction

The platforms of interest that support the branch misprediction event are the Pentium III, Itanium, Power3-II, and R12K. For these super-scalar processors, which support speculative execution, effective branch prediction is essential for good performance. The branch prediction algorithms implemented on these processors are proprietary and quite sophisticated [5], so much that reverse-engineering them is extremely difficult. Thus, rather than trying to characterize these algorithms and study various types of branching behaviors, we identify a control structure for which branch misprediction behavior is identical across platforms and exploit this observation to produce a cross-platform microbenchmark. The hypothesis is that, for all platforms of interest, a for-loop causes a branch misprediction event only upon exit and all other branches are predicted correctly. Using this hypothesis, the branch misprediction microbenchmark, shown in Figure 3, was designed. It consists of two nested for-loops. The inner loop is iterated a constant number of times, i.e., 10 times; while the outer loop is iterated a variable number of times. Scalability is achieved by parameterizing the outer loop, which is iterated the specified number of times. To cause n mispredictions, the input parameter must cause $n-1$ iterations of the outer loop, which

result in *n-1* mispredictions on the *n-1* exits of the inner loop plus one misprediction on exit from the outer loop. The initial port of the microbenchmark was to the Pentium III, where the hardware-reported count is within 1% of the predicted count. The code was ported to the remaining processors of interest, with similar (<1%) results, which makes this code our first verified cross-platform microbenchmark.

```
for (i=1; i<n; i++)          // n-1 iterations
{                            // inner loop causes 1 branch misprediction
   for (j=0; j<10; j++);     // for a total of n-1 mispredictions
}                            // plus one as outer loop exits for a total of n
```

Figure 3. Cross-platform branch misprediction microbenchmark

4.2 Instruction Cache Misses

The instruction cache (Icache) miss microbenchmark is based on the standard definition of an Icache miss, i.e., a miss occurs when a referenced instruction is not resident in the cache. It consists of an unconditional while loop that is twice as large as the Icache. Since the Icache size is processor dependent, it is a microbenchmark design parameter. The idea behind making the loop twice as large as the cache size is so that the instructions that comprise the loop are never entirely cache resident and that references to instructions in the second half of the loop cause instructions in the first half to be evicted. If the microbenchmark is designed with this specification in mind, it generates a miss on each cache line access. The loop consists of several instruction blocks, each of which is comprised of high-level instructions that generate "enough" assembler instructions to fill a cache line and produce one miss. Thus, to implement a microbenchmark with this specification, the cache line size, ISA, and instruction sizes necessarily influence the design and are, thus, microbenchmark design parameters.

To provide microbenchmark scalability, an input parameter specifies the number of Icache misses to be generated and each block of the loop contains "dummy" instructions needed to fill a cache line as well as an increment instruction to count the number of instruction blocks executed. The final instructions in each block test the count and break the loop if the specified number of blocks has been executed. The

number of Icache misses to be generated is the only input parameter to the microbenchmark; all the other parameters are design parameters.

Obviously, this microbenchmark design is portable but the implementation is not. The implementation process consists of three steps.

1. Identify the instruction size, *i*, and the number of instructions, *s*, needed to fill an Icache line.
2. Build a high-level language instruction block that generates *s* assembler/machine instructions.
3. Build a microbenchmark that contains *b* instructions blocks, where $b*s$ is two times the size of the Icache.

The initial implementation of the microbenchmark design was to the Pentium III, which has a 16KB Icache with a 32B line. Being a CISC architecture, and thus not having a fixed instruction size, the definition of a high-level language instruction block that fills a cache line required additional work. First, an initial hypothesis was made with respect to average instruction size, i.e., four bytes, and a high-level language (C) instruction block that generates eight assembler instructions, depicted in Figure 4, was written. Next, a script was used to build microbenchmarks with monitored code sections comprised of a sequence of different numbers of the instruction block, and to record the size of each resultant binary. The resulting data shows that, contrary to our hypothesis, on average, instruction size is 28% longer than 4 bytes. With this information, it was possible to write a loop twice as large as the 16KB Icache; (1/1.28) * 2 * (16384 / 32) = 800 blocks were needed and 28% less instruction blocks needed to be accessed to generate the specified number of Icache misses.

The predicted Icache miss event count is the “specified” number given by the microbenchmark input parameter. When the microbenchmark was executed, the hardware-reported event count was within 1% of the predicted event count.

// **High-level Block**	# **Assembler Block**

<pre>a = i + value;   // dummy add
i++;   // incr count
if (i==pred_miss)   // spec'd count?
break;   // stop</pre>
	<pre>movl -8880(%ebp), %eax # value
	addl -8872(%ebp), %eax # i + value
	movl %eax, -32(%ebp) # a = i + value
	incl -8872(%ebp) # i++;
	movl -8872(%ebp), %eax # i
	movl -8876(%ebp), %edx # pred_miss
	cmpl %edx, %eax # i == pred_miss
	je ..B2.830 # break (cond.)</pre>

Figure 4. Pentium III high-level instruction block and corresponding assembler code

This same microbenchmark can be used to count level-two (L2) Icache misses. When ported for this purpose to the Pentium III, similar results (<1%) were observed.

4.3 Cycles

All platforms of interest support an event that permits the counting of the number of cycles elapsed during the execution of monitored code. To be able to predict the elapsed number of cycles, however, events that may not be modeled accurately, e.g., latency-hiding events, pipeline stalls, and resource contention must be eliminated. For example, in most modern processors, multiple cache misses can be outstanding; miss handling is concurrent with instruction execution, thus hiding a portion of the miss penalty; multiple functional units execute concurrently and compete for access to resources such as caches and microarchitectures buses; and data dependencies may introduce bubbles in the pipeline. Such events can be hard to model and, thus, may make it difficult to predict event counts. Thus, the initial idea behind the cycles microbenchmark design was to monitor only one type of instruction, stressing only one functional unit, the one with the smallest latency, generating an instruction stream that could be modeled easily. For the Pentium III, the first platform to which the microbenchmark was ported, this translates into stressing the integer ALU with integer add instructions, each of which has a latency of one cycle. However, this sounds easier than it actually is.

The first hurdle that presented itself is associated with the compiler. The microbenchmarks are compiled with no optimization so that the monitored code is not changed. If the compiler changes the code organization or eliminates instructions, event count prediction is thwarted and hand-tuning of the

assembler code, when necessary, is difficult. However, with no optimization, no variables are allocated to registers, and a simple add instruction in the high-level language (C) results in three memory-accessing instructions: a load, load-add, and store, which may introduce additional cycles, especially when they generate cache misses. To eliminate these extra, unpredictable cycles, declaration of register variables is done manually. But in order to identify registers available for allocation to variables, the assembler code is analyzed to determine the liveness [5] of registers. Figure 5 presents an example of how this is done. Once enough free registers are identified, as shown in Figure 6, the three memory-addressing instruction sequence is reduced to a single register-to-register integer add assembler instruction with one-cycle latency.

The next hurdles are associated with the microarchitecture and memory hierarchy. It was determined experimentally that for each of the two ALUs, two alternating integer add instructions, each using a unique register, need to be issued. If two instructions that use the same register are dispatched consecutively to the same ALU, then bubbles may be introduced to allow time for a register update.

Originally, to avoid the complexity associated with branch instructions, the code was implemented as an inline microbenchmark, i.e., one with sequential execution flow. However, as the code size grew, monitored event counts (which monitored cycles, cache misses, and TLB misses) indicated that instruction cache and TLB misses contributed “extra” (unpredicted) cycles to the cycle event count. To minimize these “extra” cycles associated with misses in the memory hierarchy, a relatively simple for-loop microbenchmark with only integer-add instructions in the monitored loop body was implemented. Results from the branch misprediction microbenchmark, discussed in Section 4.1, indicate that penalties due to a mispredicted branch occur only once, as the loop is exited, and do not contribute much overhead. To avoid data cache misses and associated “extra” cycles, the for-loop variables also need to be assigned to registers. The cycles introduced by the loop’s increment and compare instructions are easily predicted.

```
      …
PAPI_start                # to use %ecx in the monitored region, first check if
      …                   # whatever value %ecx had in the region will be used
PAPI_stop                 # afterwards
      …
load %eax, 8000(%esp)     # since %ecx gets a new value the first time it is
                          # referenced after the monitored region, its value in
                          # the region is not live; %ecx can be freely used
```

Figure 5. Liveness analysis of registers

# Before Manual Register Allocation	# After Allocating %ecx, %edx to *a*, b
load %eax, 8088(%esp) # a add %eax, 8086(%esp) # a + b store %eax, 8088(%esp) # a = a + b	add %ecx, %edx # a += b

Figure 6. Using free registers to simplify Pentium III cycles microbenchmark

The final cycles microbenchmark, devoid of events that introduce "extra" cycles, resulted in hardware-reported cycle event counts within 1% of predicted event counts. The design was also ported to the Power4, with similar (<1%) results.

4.4 Floating-point Instructions

The relatively simple floating-point instructions microbenchmark consists of a loop with a body containing only floating-point instructions. It is designed to permit the monitoring of combinations of add, multiply, and divide floating-point instructions, and the prediction and collection of the number of floating-point instructions executed. Predicted event counts are obtained using a script that counts the number of assembler floating-point instructions in the disassembled object module; this method is necessary since, as will be discussed later, the compiler sometimes introduces "extra" floating-point instructions that are visible only in the object module.

The floating-point instructions event was studied on the Pentium II, Pentium III, Itanium, Power3-II, Power4, and R12K. For the Itanium and R12K, hardware-reported counts are within 1% of predicted counts. For the Power3-II and Power4, initially, using predictions based on the assembler code rather than the object module, reported and predicted counts differed by a factor of two; after inspection of the object module it was found that the difference was attributable to rounding instructions introduced by the

compiler [13]. For the Pentium III, hardware-reported counts are less than predicted counts, and as of yet the reason for the differences is not understood. The differences occur when multiplication of single-precision variables (of float type in C) is used alone or in combination with other floating-point instructions. An example of such an instruction sequence is given in Figure 7; the monitored code contains nine floating-point multiplications instructions but PAPI reports a count of five. The reported counts vary depending on the instruction sequence, but they are always less than the predicted counts.

// High-level C Code	# Disassembled Code
…	
float a, b, c, init_value;	
…	…
a = b * init_value;	fmuls 0xffffff58(%ebp)
b = a * init_value;	fmuls 0xffffff58(%ebp)
c = c * init_value;	fmuls 0xffffff58(%ebp)
c = a * b;	fmuls 0xffffff60(%ebp)
a = b * c;	fmuls 0xffffff5c(%ebp)
b = a * c;	fmuls 0xffffff5c(%ebp)
a = c * b;	fmuls 0xffffff60(%ebp)
c = b * b;	fmuls 0xffffff60(%ebp)
b = a * a;	fmuls 0xffffff64(%ebp)
…	…

Figure 7. Instruction sequence that causes an unexpected floating-point instruction count

4.5 Floating-point Square Root

The Power3-II and Power4 are the only platforms of interest that support the square-root event. Ported to these systems, the relatively simple floating-point square-root microbenchmark monitors a for-loop that sequentially accesses an array of floating-point numbers, computing the square root of each. In order for square root to be implemented in hardware on these platforms, the microbenchmark must be compiled with an optimization level of at least three. Predicted event counts are obtained using a script that counts the number of high-level language square-root instructions. For the Power3-II, the hardware-reported square-root event counts match the predicted counts only when more than 100 square-root instructions are computed. Similarly, for the Power4, the reported counts match the predicted counts only when more than 86 square-root instructions are executed. In general, when the reported and predicted counts do not agree, the reported count is zero. This discrepancy indicates a possible hardware or firmware error.

4.6 Instructions Issued and Completed

The instructions issued and completed microbenchmark includes a monitored a for-loop with a body of 10-20 add instructions. The predicted event counts are made from the disassembled object module. For these two events, the predicted counts are the same as the number of dynamic instructions between the PAPI directives. This microbenchmark has been used on a variety of platforms. For example, the results on the Power3-II and the Pentium III show a small (<1%) difference between predicted and reported event counts. However, not all platforms show such results. On the Itanium platform, the reported counts are consistently 17% larger than the predicted counts. This is due to no-ops introduced by the compiler to "pad" VLIWs (very long instruction words); this was discovered by inspection of the disassembled object module. Accounting for this in the predicted counts, the difference is close to 0%. A similar situation exists on the Power4, where the initial results showed a difference of up to 400%. One hypothesis for the difference is that the way instructions are packaged inside the Power4 affects the number of instructions issued. In order to keep track of the instructions in flight, groups of five instructions are formed. The groups cannot be dispatched until all resources are available [16]. For each high-level add instruction in the microbenchmark, there are a sequence of load, add, and store instructions generated in the assembler code. It seems as if the load and store instructions could not be issued in the same group because of data hazards. This would force those instructions to be issued in different groups forcing no-ops to fill the rest of the group. However, this "packaging" is done within the microarchitecture, so unlike the Itanium, no extra instructions (no-ops) are visible in the disassembled object module. In order to minimize this effect, all load and store instructions were removed by using register variables. With this new microbenchmark the results show a 2% difference between predicted and reported counts.

5. Conclusions and Future Work

As illustrated by the events discussed in this paper, the majority of hardware-reported event counts agree with predicted counts. Earlier work [7, 9, 15], which presented work in progress, indicates differences between hardware-reported and predicted event counts that were not yet understood. Many of these differences were later understood as we refined our methodology. Tables 2 and 3 present the status

of our work to date. Table 2 describes the events studied and their respective PAPI identifiers; Table 3 presents a summary of the events and platforms that have been studied to date.

As new processors are introduced into the marketplace, the representativeness of their event counts needs to be evaluated. Developing new architecture-specific microbenchmarks is too expensive. To minimize this work, architecture-independent microbenchmarks, when applicable, are essential. The goal of this research is to develop a suite of microbenchmarks that can be ported easily to any platform. For events for which the microbenchmark must be tailored to the architecture, rather than just ported or parameterized, the goal is to port the microbenchmark design, if not the implementation. The microbenchmarks presented in this paper are a first step towards this goal. Such a suite would facilitate the evaluation of performance counters and their use by application programmers to tune their codes.

PAPI Event Name	Description
PAPI_BR_MSP	Branch instructions mispredicted
PAPI_FSQ_INS	Floating point square root instructions
PAPI_FP_INS	Floating point instructions
PAPI_L1_DCM	L1 Data cache misses
PAPI_L1_ICM	L1 Instruction cache misses
PAPI_L2_DCM	L2 Data cache misses
PAPI_L2_ICM	L2 Instruction cache misses
PAPI_LD_INS	Load instructions
PAPI_SR_INS	Store instructions
PAPI_TLB_DM	Data TLB misses
PAPI_TLB_TL	Total TLB misses
PAPI_TOT_IIS	Instructions issued
PAPI_TOT_INS	Instructions completed
PAPI_TOT_CYC	Total cycles

Table 2. PAPI event names and descriptions

Some microbenchmarks, for example the instruction cache miss microbenchmark, cannot be ported across platforms because, for the same event, the event definitions differ. This research shows that event definitions need to be standardized across platforms to avoid misunderstanding and misuse of data obtained from performance counters.

For future work, the porting of the cross-platform microbenchmarks to the platforms of interest will be completed. In addition, the events associated with the Cray X1 and R16K platforms will be evaluated using the cross-platform microbenchmarks—this will test the portability of these microbenchmarks.

Acknowledgements

We wish to thank the Department of Defense, in particular, the PET program, for support of this research.

PAPI Event Name	Itanium	Pentium III	Power 3_II	Power4	R12K
PAPI_BR_MSP		X	X	N/A	P
PAPI_FSQ_INS	N/A	N/A	X	X	N/A
PAPI_FP_INS	X	X	X	X	X
PAPI_L1_DCM	X	P	X	X	X
PAPI_L1_ICM	X	X	P	N/A	X
PAPI_L2_DCM	X		N/A	N/A	X
PAPI_L2_ICM	X	X	N/A	N/A	X
PAPI_LD_INS	X	N/A	X	N/A	X
PAPI_SR_INS	X	N/A	X	N/A	X
PAPI_TLB_DM	X	N/A	N/A	X	N/A
PAPI_TLB_TL		N/A	X	X	P
PAPI_TOT_IIS		X	X	X	X
PAPI_TOT_INS	X	X	X	X	X
PAPI_TOT_CYC		X	P	P	P

Table 3. Platforms and events under study. Legend: *N/A* –Not a PAPI supported event in that specific platform, *X* –Event study already completed, *P* –Event study in progress

References

[1] S. Browne, 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*, vol. 14, no. 3, Fall 2000, pp. 189-204.

[2] L. DeRose, "The Hardware Performance Monitor Toolkit," *Proc. Euro-Par*, 2001, pp. 122-131.

[3] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters," *PADTAD Workshop, IPDPS 2003,* 2003.

[4] GNU, "GNU Binary Utilities: objdump," http://www.gnu.org/software/binutils/manual/html_chapter/ binutils_4.html.

[5] J.L.Hennessy and D.A.Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2003.

[6] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche, "Memory Profiling using Hardware Counters," *Proc. Supercomputing 2003* (SC2003), 2003.

[7] W. Korn, P. J. Teller, and G. Castillo, "Just how accurate are performance counters?," *Proc. 20th IEEE International Performance, Computing, and Communications Conference*, 2001.

[8] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer, "End-user Tools for Application Performance Analysis Using Hardware Counters," *Proc. International Conference on Parallel and Distributed Computing Systems*, 2001.

[9] M. Maxwell, P. Teller, L. Salayandia, and S. Moore, "Accuracy of Performance Monitoring Hardware," *Proc. Los Alamos Computer Science Institute Symposium* (LACSI), 2002.

[10] MIPS R12000 Reference Manual: http://www.sgi.com/processors/r12k/manual.html.

[11] S. Moore, D. Terpstra, K. London, P. Mucci, P. J. Teller, L. Salayandia, A. Bayona, and M. Nieto, "PAPI Deployment, Evaluation, and Extensions," *Proc. Users Group Conference,* 2003.

[12] J. Mueller, "Detecting DDOS and Other Security Problems," *InformIT*, http://www.informit.com/ articles/article.asp?p=169493&seqNum=3.

[13] M. Nieto, A. Bayona, L. Salayandia, P. J. Teller, and L. DeRose, *Hardware Performance Metrics and Compiler Switches: What You See Is Not Always What You Get*, tech. report, Performance Counter Assessment Team, The University of Texas at El Paso, 2004.

[14] F. Parienté, "Performance Analysis and Monitoring Using Hardware Counters," *Technical Articles & Tips*, http://developers.sun.com/solaris/articles/hardware_counters.html.

[15] L. Salayandia, *A Study of the Validity and Utility of PAPI Performance Counter Data,* master's thesis, Dept. Computer Science, The University of Texas at El Paso, 2002.

[16] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html, 2001.