RC22938 (W0310-109) October 17, 2003
Computer Science

IBM Research Report

Hardware Performance Metrics and Compiler Switches: What You See Is Not Always What You Get

Manuel Nieto, Alonso Bayona, Leonardo Salayandia, Patricia J. Teller
The University of Texas at El Paso

Luiz Derose
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Hardware Performance Metrics and Compiler Switches: What You See Is Not Always What You Get

Manuel Nieto, Alonso Bayona, Leonardo Salayandia, and Patricia J. Teller
The University of Texas at El Paso
{manueln,leonardo,pteller}@cs.utep.edu and alonsob@utep.edu

Luiz DeRose
IBM T.J. Watson Research Center
laderose@us.ibm.com

ABSTRACT

Hardware performance counters have become an invaluable asset for application performance tuning. However, since hardware counters have in general been designed for debugging hardware and not for application performance measurements, there are no standards and there is little documentation across the industry in terms of what is being counted. This lack of standards and documentation, in conjunction with the lack of understanding by users of the performance impact of certain compiler switches, can cause pitfalls and misinterpretations when using hardware counters for characterization of application performance. Thus, although hardware performance counters can provide valuable data to the application programmer, the data must be understood! In this paper we study the performance impact of various compiler options via the use of hardware performance monitor event counts on IBM's Power3 and Power4 microprocessors. In addition, we demonstrate the need for counter data calibration, and illustrate how some hardware metrics can be misleading due to incorrect interpretation.

1. Introduction

Hardware performance counters have become an invaluable asset for application performance tuning. They can provide useful data that can be used by the application programmer to identify causes of performance degradation and, thus, ways to enhance performance. However, since hardware counters have been designed, in general, for debugging hardware and not for application performance measurements, there are no standards and there is little documentation across the industry in terms of what is being counted. Hence, this lack of standards and documentation can cause pitfalls and misinterpretations when using hardware counters for characterization of application performance.

Along these lines, the effective use of compiler options is an important technique applied by application programmers to improve performance, but the selection of a "wrong" set of options can backfire and degrade application performance. As demonstrated in this paper, hardware performance monitors can be used to understand the effect of compiler options on the performance of a program or specific regions of code. In this work we study the performance impact of various compiler options via the use of hardware performance monitor event counts on IBM's Power3 and Power4 microprocessors, providing several examples of how performance counters can be used for this purpose. We also demonstrate the need for counter data calibration and exemplify how data generated by performance counters can be calibrated. In addition, we illustrate how some hardware metrics can be misleading due to incorrect interpretation.

The remainder of the paper is organized into five sections. Section 2 briefly discusses hardware performance counters. Section 3 describes the computing environment used to perform the experiments described in Section 4. Sections 4.1 through 4.4 present experiments that demonstrate how event counts can be used to evaluate the performance effect of compiler options and, in doing so, presents what may be, for some, surprising results with respect to the performance effect of compiler optimizations. Sections 4.5 and 4.6 address counter data calibration and the level-one instruction cache hit event on the Power3. Section 5 concludes the paper with a summary and a discussion of planned future work in this research area.

2. Hardware Performance Monitors

Hardware performance monitors, which are available in most modern microprocessors, are realized by a small set of registers that count and/or record information about *events,* which are occurrences of specific signals and states related to a processor's function. Example events include the occurrence of a load, cache hit, or a floating-point operation. Originally, hardware counters were designed for hardware debugging, but since the monitoring of events has a number of uses in application benchmarking, performance analysis, and optimization, hardware counters have become an invaluable asset for application performance tuning.

Event monitoring can be accomplished by either counting the number of occurrences of the event or sampling, i.e., recording information about the event every so often. The overhead of sampling is relatively low as compared to that of aggregate counting. However, there is a tradeoff between incurred overhead and accuracy. Using sampling, aggregate counts must be estimated from samples. Thus, when using sampling, programs must run for a sufficiently long time to allow the counts to converge to expected values and must have a structure that is amenable to this type of performance evaluation. Understanding the performance impact of compiler optimizations requires accuracy and, thus, aggregate counting is the sound option for this work [9]

3. Experimental Environment

The work described in this paper focuses on:

- two microprocessors families: IBM's Power3 and Power4;
- two operating systems: AIX 4.3 and AIX 5.1;
- three compilers: native C and Fortran compilers for the IBM POWER series (*XLC* Version 6.1 and *XLF* Version 7.1), and *GCC*;

- three performance monitor interfaces: PMAPI [4] IBM's native Performance Monitor Application Program Interface to the hardware counters in the Power series microprocessors; the HPM Toolkit [2] IBM's native High Performance Monitor; and PAPI [1] a cross-platform Performance Application Program Interface, which is built upon vendor interfaces, such as PMAPI; and
- aggregate event counts, rather than sampled event counts.

Microprocessors and Operating Systems:
The IBM Power4 processor used in this study is a component of Cheetah [11], **Error! Reference source not found.**a 27-node IBM pSeries system operated by the Computer Science and Mathematics Division of Oak Ridge National Laboratory. Cheetah has 32 1.3 GHz Power4 processors per node, two processors per chip. Each chip is associated with three levels of cache: on-chip level-one instruction caches (64 KB per processor), on-chip level-one data caches (32 KB per processor), an on-chip 1.5 MB level-two unified cache shared by the two on-chip processors, and an off-chip 32 MB level-three cache. The resident operating system is AIX 5.1.

Two IBM Power3-based systems running AIX 4.3, both operated by the University of Tennessee-Knoxville's Computer Science Department, also are used in this study: a two-CPU SMP (symmetric multiprocessor) comprised of 200 MHz Power3 processors and a two-CPU SMP comprised of 375 MHz Power3+ processors.

Compiler Options:
Various *XLC* and *XLF* compiler options [6] are investigated; these include:

-qrealsize=8
Sets the default size of real variables (including constants) to 8 bytes, i.e., double precision (real*8).

-qtune=pwrX, where *X* = 3 and 4 for the Power3 and Power4, respectively
Tunes the specified instruction selection, scheduling, and other implementation-dependent performance enhancements for the specified architecture.

-qarch=pwrX, where *X* = 3 and 4 for the Power3 and Power4, respectively
Controls which instructions the compiler generates and produces a binary containing instructions that run on a Power*X*, but potentially may not run on earlier versions.

-***Oi***, where *i* specifies the optimization level

Performance Monitor Interfaces:
Both the IBM Power3 and Power4 series microprocessors have eight physical registers associated with the performance monitor. These counters can be accessed by the IBM native API: PMAPI or by higher level APIs, such as the HPM Toolkit or PAPI. These two interfaces use the PMAPI at a lower level. They

differ from the PMAPI in that the PMAPI is harder to set up and use but has a lower overhead, since it is implemented at kernel level.

The HPM Toolkit supports performance data capture, analysis, and presentation for applications written in Fortran, C, and C++, executing on sequential or parallel systems, running shared-memory, message-passing, or mixed-paradigm applications. In addition to presenting the raw counter data, the HPM Toolkit also computes a rich set of derived metrics, which are dependent on the hardware events being counted. PAPI is a cross-platform interface that provides access to all native counter events and counting modes. It is intended for use by application engineers and tool developers, also providing an easy-to-use interface for starting, reading, and stopping a specified list of event counters and derived metrics.

4. Evaluation of Performance Effects of Compiler Options, Calibration, and Interpretation

The following investigations of the performance effects of compiler options demonstrate (1) how event counts generated by a hardware performance monitor can identify causes of performance degradation and can assist the programmer in tuning the performance of application code, (2) how event counts can be calibrated, and (3) how important it is to understand the definition of an event. Some of these investigations use real benchmarks for these purposes, while others (those discussed in Section 4.3-4.6) utilize micro-benchmarks, designed specifically to stress a particular portion of the microarchitecture or memory hierarchy and the associated events. In the latter case, the methodology used to study the data generated by performance counters is similar to that used in [7, 8, 10, and 13]. It consists of seven phases, which are repeated as necessary. For a specific event, the seven phases are as follows:

1. **Micro-benchmark**: Design and implement a validation micro-benchmark that permits event count prediction.
2. **Prediction**: Predict event count using tools and/or mathematical models.
3. **Data collection-1**: Collect hardware-reported event count data using a high-level API such as PAPI or HPM.
4. **Data collection-2**: Collect predicted event count data using a simulator such as SIGMA [3] (not always necessary or possible).
5. **Comparison**: Compare predicted and hardware-reported event counts.
6. **Analysis**: Analyze results to identify and possibly quantify differences.
7. **Alternate approach**: When analysis indicates that prediction is not possible, use an alternate approach to either verify reported event count accuracy or demonstrate that the reported event count seems reasonable.

Micro-benchmarks: A *validation micro-benchmark* is a simple program, usually small in size, designed to permit prediction of a particular event count. It is based on a section of code that is being monitored; the monitored code is delineated by calls to the performance monitor interface that set up the performance counters to monitor the target event and to start, stop, and read the counters. A micro-benchmark's size, simplicity, or execution time facilitates the tracing of its execution path and/or prediction of the number of times the target event is generated. 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 other processes, including operating system processes. This is important because even though an event count generated on behalf of a process under study is differentiated from the event counts of other processes, for some events the event count of the process under study may be severely affected by the processing environment. For example, if the execution of monitored code spans multiple time slices, then hardware-reported counts associated with memory hierarchy events may be perturbed by the memory activity of other processes.

Data Collection: A test suite is executed for any given event. The test suite is comprised of different versions of the event's validation micro-benchmark, which differ with respect to predicted event count. In general, a test suite is comprised of benchmark versions that generate from 1 to 1,000,000 instances of the event. For some events, due to platform limitations or benchmark design, a smaller test suite is used. Each version of the benchmark is executed 100 times and the mean and standard deviation of the hardware-reported counts are computed. Using an average takes into account the variability of the reported counts. The standard deviation as well as data inspection ensures the stability of the results and identifies data anomalies that call for further study. A script, as opposed to wrapping the micro-benchmark in a for-loop, is used to run 100 instances of a benchmark. The latter would cause reuse of benchmark and interface data as well as instructions and, consequently, could eliminate events that would be generated otherwise or introduce some events that would not be generated otherwise. For example, using the latter, instruction cache misses may only occur during the execution of the first instance.

4.1 Performance Degradation Due to D0 Constants and –qrealsize=8

The study of the effect of the *XLF* Fortran compiler option *–qrealsize=8* on the performance of the SPEC 2000 Swim benchmark [12] using the HPM Toolkit, demonstrates how the definition of a constant and compiler options, without an understanding of their performance effects, can result in significant performance degradation. Swim is a two-dimensional simulation model of the dynamics of the shallow water wave equations [14] Originally, it was written in Fortran 77, at the National Center for Atmospheric Research (NCAR), for performance analysis of supercomputers. From the Swim benchmark the "DO

`100`” loop, depicted in Figure 1, is of particular interest with respect to the identified performance degradation. This `DO` loop computes expressions with variables and constants such as:

```
CV(I,J+1)  = .5D0 * (P(I,J+1) + P(I,J)) * V(I,J+1)
```

Line #	**Fortran Code**
278	`DO 100 J = 1, N`
279	`DO 100 I = 1, M`
280	`      CU(I+1,J)  = .5D0 * (P(I+1,J) + P(I,J)) * U(I+1,J)`
281	`      CV(I,J+1)  = .5D0 * (P(I,J+1) + P(I,J)) * V(I,J+1)`
282	`      Z(I+1,J+1) = (FSDX * (V(I+1,J+1) - V(I,J+1)) - FSDY *`
283	`1       (U(I+1,J+1) - U(I+1,J))) / (P(I,J) + P(I+1,J) + P(I+1,J+1) +`
284	`2       P(I,J+1))`
285	`      H(I,J)     = P(I,J) + .25D0 * (U(I+1,J) * U(I+1,J) +`
286	`1       U(I,J) * U(I,J) + V(I,J+1) * V(I,J+1) + V(I,J) * V(I,J))`
	`100   CONTINUE`

Figure 1. Swim DO 100 Loop

From the example above, “`.5D0`” represents a double-precision constant (eight bytes); the other variables are elements of double-precision two-dimensional arrays. When the benchmark is compiled with the *–qrealsize=8* compiler option, the benchmark’s performance degrades. As shown below, the degradation is due to the “`D0`” constants coupled with the *–qrealsize=8* option.

To determine the cause of performance degradation, two sets of experiments were conducted. The first set of experiments consists of compiling the benchmark on the IBM Power3 in two different ways (one with *–qrealsize=8* and one without), producing two different codes:

SwimR8D0: *–O3 –qtune=pwr3 –qarch=pwr3 –qrealsize=8*
SwimD0 : *–O3 –qtune=pwr3 –qarch=pwr3* (no –qrealsize=8)

The second set consists of two different codes, **SwimR8E0** and **SwimE0**, produced by changing all “`D0`” constants in the source code to “`E0`” (single-precision, four-bytes) constants and compiling using the same two different compiler options specified above. Table 1 shows the performance data collected for the “`DO 100`” loop when running 120 iterations on the Power3 and Power4 systems, with `N=M=512`. The events monitored with the HPM library are loads completed (*PM_LD_CMPL*), stores completed (*PM_ST_CMPL*), instructions completed (*PM_INST_CMPL*), cycles (*PM_CYC*), and execution time. From the performance data in Table 1, it is evident that the number of instructions and the number of loads completed are causing the performance of SwimR8D0 to be significantly less than the other three code versions. Concentrating on the effect of "`D0`" constants and the *–qrealsize=8* compiler option, for SwimR8D0 the number of loads completed is almost double.

Code Version	SwimD0	SwimR8D0	SwimE0	SwimR8E0
Constants	D0		E0	
Compiled with	*–O3*	*-qrealsize=8 -O3*	*–O3*	*-qrealsize=8 -O3*
PM_LD_CMPL	345,785,040.00	629,269,080.00	345,785,040.00	345,785,040.00
PM_ST_CMPL	172,646,640.00	125,829,360.00	172,646,640.00	172,646,640.00
PM_INST_CMPL	1,110,347,639.00	3,996,245,279.00	1,110,347,639.00	1,110,347,639.00
PM_CYC	1,159,037,215.00	4,103,932,982.00	1,159,847,577.00	1,160,975,812.00
Execution time	5.80 Sec.	20.52 Sec.	5.80 Sec.	5.80 Sec.

Table 1. Performance Metrics: Do 100 Loop on Power3

The performance degradation experienced by SwimR8D0 is due to the fact that when the compiler option *-qrealsize=8* is used, "`D0`" constants are promoted to 16 bytes. 16-byte constants do not fit in a (64-bit) register. As a result, every time a 16-byte constant is referenced it is loaded from memory, causing the number of loads completed to increase, and hence performance to degrade.

The promotion of "`D0`" constants to 16-byte constants is documented. However, this result indicates that a lack of understanding on the part of the application programmer about the side effects of certain compiler options can lead to significant loss of performance. The programmer may use "`D0`" constants and the compiler option *-qrealsize=8* to increase computational precision, not knowing that this choice will result in a significant loss of performance.

4.2 Performance Degradation Due to Register Spills Associated with –qrealsize=8

The study of the effect of the *XL* Fortran compiler option *–qrealsize=8* on the performance of the Swim benchmark, using the HPM Toolkit, also demonstrates how register spills and the related loads and stores generated by a compiler option can result in significant performance degradation. For this investigation SwimD0 and SwimR8D0 are used (see Section 4.1).

The code generated for SwimR8D0 contains a significantly smaller number of assembler instructions than the code generated for SwimD0, but looking at Table 1, we observe that it executes approximately 3.6 times more instructions. By inspecting each assembler code, we observe that SwimR8D0 includes library functions to compute operations on 16-byte values. For instance, a multiplication in line 280 of the Fortran source code, i.e., `.5D0 * (P(I+1,J) + P(I,J)) * U(I+1,J),` is computed with the function "*`_xlqmul,`*" which is not the case for SwimD0. The number of floating-point loads completed for SwimR8D0 is almost double that of SwimD0. This is due to library function calls such as "`_xlqmul`", which due to their nature are probably slow and require several loads in order to perform computations on 16-byte operands. Consequently, this increases the execution time of SwimR8D0, making it 3.5 times slower than SwimD0. Note, however, that the number of stores for SwimD0 is greater than that of SwimR8D0. This is because of register spills triggered by not having enough registers to store all needed values; some values are stored onto the stack to free registers. On the IBM systems, it is the responsibility of the called function, e.g. "`_xlqmul`", to save and restore some registers. Hence, there are no register spills for SwimR8D0 and the number of hardware-reported stores is almost exactly the number of stores expected from inspection of the source code (about four stores per iteration).

4.3 Performance Degradation Due to Square Root Implementation

The performance effect of compiler options often can be quantified by examining event counts via PAPI and the HPM Toolkit. For example, the *Total Floating Point Instructions* event count on IBM's Power3 and Power4 processors can be used to quantify the performance effect of compiler options affecting square root implementation. The Power3 and Power4 can perform square root (SQRT) operations in hardware, which is significantly faster than software implementations that use a sequence of other, more common, floating-point instructions to implement iterative algorithms that approximate SQRT (e.g., Newton-Raphson). Whether the hardware or software implementation is utilized depends on the compiler options specified. The native C compiler on the Power platform, *XLC*, requires the *qarch=pwrX* (where *X* = 3 or 4) and *O3* flags to be used in order to execute SQRT instructions in hardware. The *qarch* flag is used to specify either a common ISA for backward-compatibility support among Power platforms or to set an ISA for a specific platform. For example, setting *qarch=pwr3* results in executable code specifically targeted at the Power3 platform. According to the *XLC* compiler documentation, the *O3* flag (in combination with the *qarch* flag set to the specific Power platform) causes the compiler to generate code that includes hardware-implemented SQRT instructions, rather than calls to a library routine that implements SQRT in software.

The Power platforms have two native events that are triggered by a SQRT instruction executed via hardware: one counts SQRT operations executed in hardware (*PM_FSQRT*) and the other counts floating-point instructions completed (*PM_FPU0_CMPL*, *PM_FPU1_CMPL*). To expose the differences in the PAPI "floating-point instructions" event (*PAPI_FP_INS*) and associated performance due to compiler options, a micro-benchmark that monitors the number of floating-point (FP) instructions executed (*PAPI_FP_INS*), FMA (fused multiply-add) instructions completed (*PAPI_FMA_INS*), total cycles (*PAPI_TOT_CYC*), and FLOPS (derived from the number of cycles and the number of FP instructions executed) was designed and implemented. The monitored code is shown in Figure 2. It uses the SQRT operation in conjunction with a FP multiplication (which appears to be needed to trigger the *PM_FSQRT* event on the Power3[1]).

```
double zz[100];
//Setup and initialization code …
[Library call to either PAPI or HPM Toolkit to start the counters]
   /* monitored code */
   for (i=0;i<100;i++) zz[i] = 1.1*sqrt(zz[i]);
 [Library call to either PAPI or HPM Toolkit to stop the counters]
```

Figure 2. SQRT (Loop) Micro-benchmark for the Power3

On the Power3 the test suite is comprised of different versions of this benchmark. At the high level, they differ with respect to the number of iterations executed and, thus, the number of events generated. At the low level, they differ with respect to their compilation: one compilation causes SQRT instructions to be executed in hardware and the other causes them to be executed in software. The first uses the *03* optimization level and the Power3 ISA (*qarch=pwr3*) to force the use of hardware SQRT instructions; the latter also uses the *O3* optimization level, but uses the common Power ISA (i.e., the *qarch* flag not set) and links to the default math library included in AIX 4.3 to implement SQRT instructions. Tables 2 and 3 report the results of running these benchmarks with PAPI library calls. On the Power4 the test suite is the same except for the addition of one more compilation, which produces code using the Power4 ISA by setting *qarch* to *pwr4*. These results, for benchmarks with both PAPI and HPM library calls, are shown in Tables 4, 5, and 6 (HPM is used to count the FP store event which is needed to accurately compute MFlops/Sec on the Power4).

Table 2 shows that the reported number of FP instructions executed when SQRT is performed on the Power3 by hardware is about double the number of iterations, which indicates that the event count accurately accounts for the SQRT and the multiply instructions. Notice that in this case the multiply operation is implemented via an FMA instruction, which is reflected in the corresponding FMA event count. As for SQRT operations implemented on the Power3 in software, Table 3 shows an increase in the FP instructions, as compared to the hardware implementation, by a factor of 20. This is because each SQRT instruction is implemented by 20 FP instructions, out of which 14 are FMA instructions.

Iterations	**100**	**1000**	**10,000**	**100,000**	**1,000,000**
Mean FMA	100.0	1,000.0	10,000.0	100,001.4	1,000,012.8
Mean FP	200.0	2,002.0	20,002.1	200,005.9	2,000,038.4

[1] On the Power3 the PAPI SQRT event (*PAPI_FSQ_INS*) is not always triggered accurately. It was found through experimentation that the event is stable when a SQRT instruction is paired with a multiply instruction.

Mean Cycles	1,639.6	12,369.4	118,879.0	1,185,959.3	11,940,171.7
Mean MFLOPs/sec	24.40	32.37	33.65	33.73	33.50

Table 2. SQRT Micro-benchmark—Hardware-implemented SQRT Instructions—Power3

SQRT instrs	**100**	**1000**	**10,000**	**100,000**	**1,000,000**
Mean FMA	1,400.0	14,000.0	140,000.6	1,400,003.8	14,000,039.1
Mean FP	2,101.0	21,001.9	210,003.8	2,100,009.5	21,000,077.0
Mean Cycles	6,910.6	63,190.4	626,757.7	6,264,917.9	62,707,742.9
Mean MFLOPs/sec	60.81	66.47	67.01	67.04	66.98

Table 3. SQRT Micro-benchmark—Software-implemented SQRT Instructions—Power3

The difference in execution times on the Power3 indicates about a 5x performance improvement when SQRT instructions are implemented in hardware, rather than software. However, a misleading metric, in this case, is MFLOPs/sec: although the hardware implementation results in a better (smaller) execution time than the software implementation, the associated MFLOPs/sec rating is worse (smaller).

Iterations	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Mean SQRT	0.00	0.00	0.00	0.00	0.00
Mean FMA	1,241.12	12,409.00	124,021.00	1,240,135.50	12,400,064.00
Mean FP	2,161.97	21,608.50	216,014.00	2,160,138.00	21,600,272.00
Mean FP stores	200.00	2,000.00	20,000.00	200,000.00	2,000,000.00
Mean Cycles	14,334.08	129,022.50	1,274,581.00	13,075,803.00	131,143,013.00
Mean MFLOPs/sec	196.07	217.72	220.32	214.76	214.120

Table 4. SQRT Micro-benchmark—Software-implemented SQRT Instructions—Power4

Tables 4, 5, and 6 show the results, including data for the SQRT and FP store event counts, for the SQRT micro-benchmark executed on the Power4. The FP store event is monitored by the HPM Toolkit, not PAPI. This event is very important when evaluating FP counts on the Power4 because it was proven by experimentation, as well as confirmed by the vendor, that this event is included in the total FP instruction count, which is not the case on the Power3.

The difference in execution times on the Power4 of the software- and hardware-implemented SQRT benchmarks increases with the number of iterations. The benchmark with *qarch=pwr3* executes about 5x faster. In this case, the MFLOPs/sec rate is again misleading. First, the MFLOPs/sec rate for the software-implemented SQRT benchmark is higher than that of the *qarch=pwr3* benchmark even though it consumes a considerably larger number of cycles. Second, comparing the execution times and MFLOPs/sec rates of the software- and hardware-implemented benchmarks with *qarch=pwr4*, the difference in execution times grows while the MFLOPs/sec rates of the hardware-implemented benchmark overtakes that of the software-implemented benchmark after 1000 iterations but only by .4%, while the execution time is about 650% smaller.

Iterations	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Mean SQRT	100.00	1,000.00	10,000.00	100,000.50	1,000,003.50
Mean FMA	0.00	0.00	0.00	0.00	0.00
Mean FP	300.43	3,000.00	30,001.00	300,000.50	3,000,012.50
Mean FP stores	100.00	1,000.00	10,000.00	100,000.00	1,000,000.00
Mean Cycles	3,562.34	26,831.00	240,400.00	2,407,495.00	24,586,506.00
Mean MFLOPs/sec	109.63	145.35	162.23	161.99	158.62

Table 5. SQRT Micro-benchmark—Hardware-implemented SQRT Instructions—Power4 (*qarch=pwr3*)

Iterations	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Mean SQRT	100.00	1,000.00	10,000.00	100,000.00	1,000,005.50
Mean FMA	0.00	0.00	0.00	0.00	0.00
Mean FP	300.00	3,000.00	30,000.00	300,000.00	3,000,013.50
Mean FP stores	100.00	1,000.00	10,000.00	100,000.00	1,000,000.00
Mean Cycles	3,105.58	18,982.50	170,126.00	1,720,597.00	17,877,625.50
Mean MFLOPs/sec	125.58	205.45	229.24	226.66	218.15

Table 6. SQRT Micro-benchmark --Hardware-implemented SQRT Instructions—Power4 (qarch=pwr4)

Tables 5 and 6 also show that on the Power4 the PAPI SQRT event gives the correct number of SQRT instructions[1] and that instead of the expected 200 total FP instructions executed (100 SQRTs and 100 multiplications) there are actually 300. This is because store instructions are included in the FP instructions executed event count.

4.4 Performance Degradation Due to Rounding Instructions

The following examples demonstrate how performance counters, accessed through PAPI, can be used to understand why one version of a program, which uses single-precision floating-point variables, has a better MFLOPs/sec rate but poorer performance with respect to execution time than another version of a program, which is compiled differently. A set of micro-benchmarks, written in C, was designed and implemented to study the PAPI floating-point instructions executed (*PAPI_FP_INS*) event. The monitored code of each benchmark is comprised of only one type of floating-point instruction: add, multiply, or FMA (fused multiply-add instructions that combine multiply and add operations without an intermediate rounding operation [5]). Each benchmark was subdivided into single-precision (32-bits) and double precision (64-bits) operands. The monitored code consists of a for-loop with 50 instructions, (10 replications of a template shown as an example in Figure 3). Each of these micro-benchmarks represents a set of benchmark programs that differ at the high level only in granularity, i.e., the number of iterations of the for-loop executed and, thus, the number of FP instructions executed, and at the low level in how they are compiled (*XLC* with and without the compiler option *–qarch=pwr3, and GCC* with no flags).

```
a = init_value;
b = init_value;
c = init_value;
a = b + init_value;
b = a + init_value;
c = a + b;
a = b + c;
b = a + c;
```

Figure 3. Template Code Section from Floating-point Addition Micro-benchmark

4.4.1 Floating-point Additions – Power3

Table 7 shows the HPM Toolkit results for the single-precision FP addition benchmarks on the Power3+. The expected number of floating-point instructions was estimated from the source code, rather than the assembler code. For the *XLC –qarch=pwr3* compilation the predicted and hardware-reported counts for the floating-point instructions executed are essentially identical. In contrast, for both the *GCC with no flags* and *XLC with no flags* compilations the hardware-reported counts are approximately twice the predicted counts. This multiplicative difference is due to rounding instructions generated by the compiler. The Power3+ performs all FP operations in double precision, i.e., the operands fed into the FP unit are comprised of 64 bits. Accordingly, in the case of the *GCC with no flags* and *XLC with no flags* compilations, the addition is performed by the `fa` assembler instruction, which produces a double-precision result; since the target variable for these benchmarks is single precision, before storing the result to memory, it must be converted to single precision. This is not the case for the *XLC -qarch=pwr3* compilation because the FP addition is implemented by the `fadds` assembler instruction, instead of the *fa* or equivalent `fadd` assembler instruction. The `fadds` instruction produces a single-precision result; thus, rounding (the `frsp` (FP round to single precision) instruction) is not necessary. These results demonstrate the effects of compiler optimizations on event counts.

Of course, rounding is not necessary when double-precision operands are employed and the result is stored at a memory location associated with a double-precision variable. Hence, the hardware counts for the double-precision FP addition micro-benchmarks, regardless of compiler, agree with the predicted.

Single-precision Floating-point Addition Micro-benchmark							
GCC no flags							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
Mean-Reported Count	100	1,000	10,000	100,000.1	1,000,001	10,000,007	100,000,073
Standard Deviation	0	0	0.1414	0.2398	0.7669	2.1093	7.8350
Predicted Count	50	500	5,000	50,000	500,000	5,000,000	50,000,000
XLC no flags							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
Mean-Reported Count	100	1,000	10,000	100,000	1,000,000	10,000,005	100,000,042

Standard Deviation	0	0	0	0.100502	0.470087	2.236068	5.827450873
Predicted Count	50	500	5,000	50,000	500,000	5,000,000	50,000,000
XLC –qarch=pwr3							
Iterations	1	10	100	1000	10000	100000	1000000
Mean-Reported Count	50	500	5,000	50,000.04	500,000.3	5,000,004	50,000,050.6
Standard Deviation	0	0	0	0.244081	0.477282	2.066497	33.52976138
Predicted Count	50	500	5,000	50,000	500,000	5,000,000	50,000,000

Table 7. Floating Point Instructions Executed—Power3+

4.4.2 Fused Multiply-Add (FMA) – Power3

A similar situation to that discussed in Section 4.3 is associated with FMAs. Due to the use of rounding instructions to implement FMAs on single-precision variables, one version of a program has a better MFLOPs/sec rate but poorer performance with respect to execution time than another version of the program, which is compiled differently. As shown in Table 8, for the single-precision FMA micro-benchmarks compiled with *XLC –qarch=pwr3*, the hardware-reported counts accessed via PAPI and expected counts are almost identical. As in the case of FP addition (discussed above), the compiler uses an instruction, the *fmadds* (FP multiply-add single), that produces a single-precision result, thus, requiring no rounding. In contrast, for either *GCC with no flags* or *XLC with no flags*, the hardware-reported event counts are significantly larger than the expected counts; in the case of *GCC*, the hardware-reported counts are about three times larger and in the case of *XLC* with no flags, twice as large. The reason for this is twofold. In both cases a rounding instruction, in this case a *frsp* instruction, is needed since the arithmetic is done in double precision but the result is stored in single precision. Additionally, in the case of the *GCC* compilations an FMA in C generates one addition and one multiplication assembler instruction (rather than an FMA instruction), as well as the *frsp* assembler instruction.

Single-precision FMA Micro-benchmark—Power3+							
GCC							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
Mean	150	1,500	15,000	150,000.1	1,500,002	15,000,019	150,000,184
Standard Deviation	0	0	0	0.322326	1.155069	4.620275	11.76643445
Expected Count	50	500	5,000	50,000	500,000	5,000,000	50,000,000
XLC (no flags)							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
Mean	100	1,000	10,000	100,000	1,000,001	10,000,007	100,000,074
Standard Deviation	0	0	0	0.244081	0.557446	2.695423	7.494215456
Expected Count	50	500	5,000	50,000	500,000	5,000,000	50,000,000
XLC –qarch=pwr3							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
Mean	50	500	5,000.0101	50,000.01	500,000.4	5,000,006	50,000,062.04
Standard Deviation	0	0	0.1005038	0.100504	0.553463	2.130919	6.12788874
Expected Count	50	500	5,000	50,000	500,000	5,000,000	50,000,000

Table 8. Floating-point Instructions Executed — Single-precision Multiply-add Micro-benchmarks—Power3+

In summary, the *XLC with no flags* and *GCC with no flags* compilations produce two and three times as many FP instructions, respectively, as the *XLC –qarch=pwr3* compilation to execute the same code section. Does this mean that by using somewhat inferior code the MFLOPs/sec rate can be increased? This would be the case if the rounding instruction were inexpensive with respect to cycles. To answer this question we monitored the number of cycles associated with the execution of the FP addition code produced by each compilation. As Figure 4 illustrates, comparing the single-precision benchmark results, the *XLC –qarch=pwr3* compilation produces the worst MFLOPs/sec rate, while the *XLC no flags* compilation produces the best even though its execution time is significantly greater. More important is the fact that the *XLC*/single-precision has the greatest MFLOPs/sec rate, even though it affords less precision and has the greatest execution time. In addition, as Table 9 shows, since the `frsp` instructions in the *GCC*/single-precision compilation are inexpensive in terms of cycles, its MFLOPs/sec rate is higher than that of the *GCC*/double-precision compilation, which generates improved precision and executes in about the same amount of time.

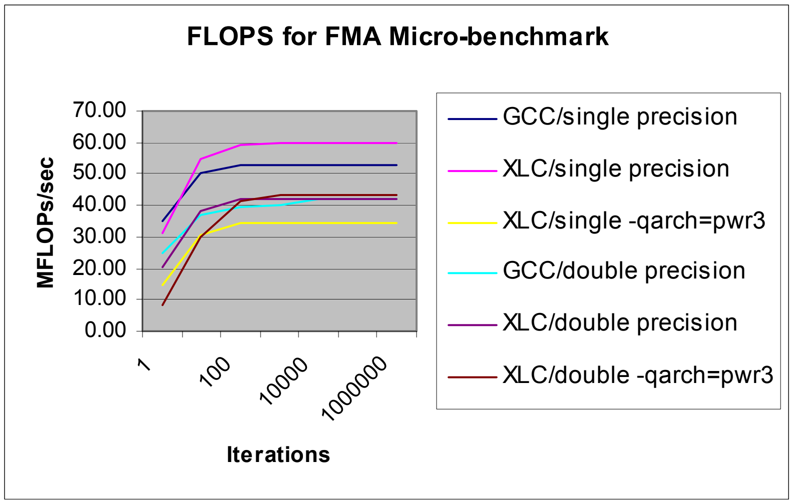

Figure 4. MFLOPs/sec Rating for FMA Micro-benchmark

FMA Micro-benchmark - Power3+

GCC single precision							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
FP ins	100.00	1,000.00	10,000.02	100,000.06	1,000,000.73	10,000,006.80	100,000,072.97
Cycles	1,597.66	11,090.55	106,441.55	1,059,742.69	10,590,825.65	105,902,284.13	1,059,016,381.89
MFLOPs/sec	23.47	33.81	35.23	35.38	35.40	35.41	35.41

GCC double precision							
Iterations	1	10	100	1,000	10,000	100,000	1,000,000
FP ins	50.00	500.00	5,000.00	50,000.00	500,000.38	5,000,004.65	50,000,048.69
Cycles	1,520.86	10,065.82	94,552.04	939,587.10	8,930,967.21	89,302,191.10	893,014,694.01
MFLOPs/sec	12.32	18,62	19.83	19.95	20.99	20.99	20.99

Table 9. Floating-point Instructions Executed, Total Number of Cycles, and MFLOPs/sec– FMA Micro-benchmark – Power3+

4.5 Event Count Calibration–Instructions Completed

Several questions have arisen concerning discrepancies with respect to hardware-reported performance data. Accuracy problems arise when the overhead, in terms of the number of generated events, introduced by the counter interface dominates an event count or contributes significantly to an event count [7] This overhead is also known as "measurement error". Problems also exist with accurately attributing hardware events to individual instructions in out-of-order processors. Architecture manuals frequently include cautions in the discussion of hardware counters, warning that the counters may not be entirely accurate and are intended only as guides to performance tuning. These and other related issues need to be investigated and documented so that application developers can use hardware performance data with confidence and not be misled by inaccurate data.

To determine the measurement error associated with the total instructions completed PAPI event, an in-line micro-benchmark was designed and developed that is based on monitored code comprised of 10 high-level (C programming language) instructions that initialize three integer variables and then perform interdependent addition operations. When compiled on the IBM Power3, the block of instructions translates into 34 assembler instructions. Different versions of the benchmark vary in the number of in-line 10-instruction blocks included in the monitored code; the baseline version has zero blocks, while the largest version has 10,000 in-line blocks (i.e., 100,000 high-level instructions). From one version to another, the number of blocks grows by a factor of 10.

Table 10 shows the results obtained from running these benchmarks on the IBM Power3. The direct count method was used to define predicted counts. As shown, for any particular version of the benchmark, all reported counts of the 100 runs are identical (the standard deviation is zero). Note that the baseline case of zero instructions gives a count of 139 instructions and the predicted and reported counts, in all cases, differ by 139. Thus, the measurement error for this event: 139, which is introduced by the PAPI routines to start, stop, and read the counters, can be used for calibration of the *Total Instructions Completed* PAPI event.

Number of C-level instructions	0 (base)	10	100	1000	10000	100000
Predicted Count	0.00	34.00	340.00	3,400.00	34,000.00	340,000.00
Mean Reported Count	139.00	173.00	479.00	3,539.00	34,139.00	340,139.00
Standard Deviation	0.00	0.00	0.00	0.00	0.00	0.00
Reported – Predicted	139.00	139.00	139.00	139.00	139.00	139.00

Table 10. Total Instructions Completed—Power3

4.6 Instruction Cache Hits–Beware of Definition–A Better Cache Hit Rate May Not Mean Better Performance!

The following investigation, which actually started as a study of instruction prefetching, illustrates how the lack of the exact definition or an incorrect interpretation of the definition of an event can mislead a user and result in misinterpretation of performance data for the event. In particular, it demonstrates how an unexpected definition of the level-one (L1) instruction cache (Icache) hit event can result in a lower hit rate and higher execution time! This can happen because on the Power3 an L1-Icache hit event is triggered when a block of instructions is fetched from the Icache into the instruction buffer, as opposed to an instruction is found in the cache.

The Power3 has a 32-KB L1 Icache that is 128-way set associative and has 128-byte cache lines. Since Power3 instructions are four bytes long, a cache line stores 32 instructions. The Power3 fetches blocks of up to eight instructions into the instruction buffer per cycle. Up to four instructions are dispatched per cycle and up to eight instructions can be executing at any given cycle. [11] To exemplify how misinterpretation of the L1-Icache hit event count can mislead a user, consider a sequential benchmark (i.e., one with no instructions that change the flow of control) that it is known to execute 480 assembler instructions. Using the generally accepted definition of a cache miss, assuming L1-Icache line of 32 instructions, and not considering prefetching, 15 misses should be generated, all of which are compulsory misses, one for each cache line. Accordingly, using the generally accepted definition of a cache hit, there should be 465 L1-Icache hits. Using performance counters to determine the actual number of L1-Icache hits, only 59 are reported. This represents around 12% of the expected number of L1-Icache hits!

This is because, as stated above, for the Power3, the expected count cannot be based on the generally accepted definition of a cache hit. Instead, it must be viewed in terms of instruction blocks fetched into the instruction buffer. Providing a good estimate, however, is rather difficult. For instance, depending on the characteristics of an application, the Power3 may not be able to fetch a block of eight instructions and dispatch four instructions per cycle. Furthermore, depending on the instruction mix, the Power3 may not be able to execute eight instructions per cycle either. As a result, the size of the instruction block fetched from the Icache into the instruction buffer per cycle may vary throughout the execution of a program.

To study this event, we ran 14 benchmarks that use performance counters, accessed via PMAPI, to monitor three events: instructions completed, instruction cache hits, and instruction cache misses. The only difference between the benchmarks is the number of instructions executed—from 32 to 262,144 instructions, increased by powers of two. Each benchmark is run 100 times to obtain the mean, standard deviation, minimum, maximum, mode and variance for each of the monitored events.

The relevant results of running these benchmarks appear in Table 11; the metrics reported for each event are expected event count (Expected), mean hardware-reported event count (Counted), and standard deviation (Std Dev). As discussed in Section 4.5, a calibration factor has been defined for the PMAPI instructions completed event. Subtracting this calibration factor from the hardware-reported counts, results in agreement with the expected counts for instructions completed. This approach cannot be used for the cache-related events (e.g., Icache hits and Icache misses), since the cache state that exists at the time monitoring begins will not be the same and, thus, the number of cache events related to measurement error will be application dependent and, thus, will vary.

	Hits			Misses			Prefetching	
Instructions	Expected	Counted	Std Dev.	Expected	Counted	Std Dev.	Counted	Std Dev.
32	3	4.32	6.58	1	2.65	3.90	0.76	1.42
64	6	9.58	7.43	2	3.63	3.37	1.83	1.42
128	12	14.69	6.30	4	3.82	3.78	3.30	1.38
256	24	30.86	7.09	8	8.51	3.32	6.03	3.27
512	48	65.51	7.27	16	16.92	3.24	14.89	1.71
1024	96	121.27	11.39	32	32.26	3.35	27.45	4.93
2048	192	240.53	22.08	64	65.26	4.35	54.02	11.90
4096	384	492.03	22.65	128	132.03	4.68	114.81	9.20
8192	768	936.47	44.65	256	261.21	8.28	201.14	23.66
16384	1536	1947.73	55.88	512	524.49	15.88	448.58	25.50
32768	3072	3866.57	87.86	1024	1039.84	20.26	867.92	45.99
65536	6144	6815.45	237.11	2048	2039.39	5.52	1113.52	154.66
131072	12288	13447.5	342.20	4096	4074.35	8.19	2098.68	229.32
262144	24576	27905.88	791.33	8192	8140.53	20.17	5070.47	500.31

Table 11. Instructions Completed, L1-Icache Hits and Misses, and Prefetching for the Micro-benchmarks

For the subject benchmarks, since the code prior to the monitored code is the same, we assume the same cache state at the time that monitoring begins and, thus, assume that the measurement error with respect to cache events is the same for all benchmarks. The measurement error associated with Icache hits and Icache misses is around 94 hits and 11 misses, respectively.

Results in Table 11 show that the expected number of L1-Icache hits is underestimated for all benchmarks. Figure 5 shows the percent difference between the expected number of L1-Icache hits and the mean of the hardware-reported L1-Icache hits. The first two benchmarks, 32 and 64 instructions, show a large difference: 30.56% and 37.37%, respectively. For the benchmarks with 128, 256, 512, 1024, 2048, 4096, 8192, 16384, and 32768 instructions the difference drops to the range 17.99-26.73%. For the last three benchmarks, 65536, 131072, and 262144 instructions, the difference drops again significantly to 9.85%, 8.62%, and 11.93%, respectively.

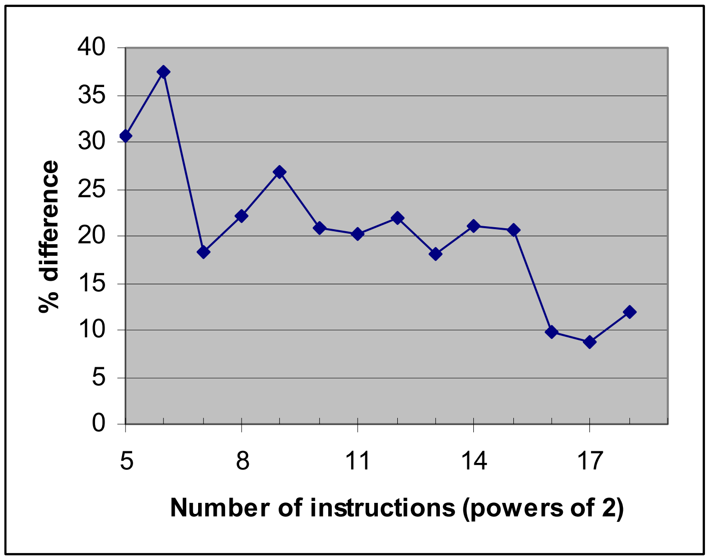

Figure 5. Percent Difference Between Expected Count of L1-Icache Hits and Mean of Hardware-reported L1-Icache Hits

A likely reason for the difference between expected and hardware-reported L1-Icache hits is that that the assumption that eight instructions are fetched into the instruction buffer does not hold throughout the execution of the benchmark. For instance, two instructions per cycle may be fetched into the instructions buffer for some section of code, while five instructions per cycle are fetched for another section of code. Because fewer instructions per cycle may be fetched into the instruction buffer than assumed for these particular benchmarks, more instruction-block fetches, from a cache line to the instruction buffer, are being performed. Consequently, the number of L1-Icache hits increases. One would think this is good! However, a greater L1-Icache hit rate than expected is not necessarily an indication of good performance. Like in these particular benchmarks, the number of instruction-block fetches from the instruction buffer is greater than expected and, consequently, more CPU cycles are used to move smaller blocks of instructions between the Icache and the instruction buffer.

Ultimately, what causes an instruction block fetched into the instruction buffer to be less than eight instructions is a combination of the number of available slots in the instruction buffer, the instruction mix, and the functional units available to execute issued instructions. The benchmarks used for these experiments execute instruction blocks similar to that shown in Figure 6. For instance, at cycle *n* two of these instruction blocks (eight instructions) are fetched into the instruction buffer. At cycle *n* + 1, four instructions are issued, leaving room to fetch another block of four instructions into the instruction buffer. At cycle *n* + 2, a four-instruction block is fetched into the instruction buffer, the 4-instruction block issued at cycle *n* + 1 starts executing, and the second four-instruction block fetched at cycle *n* is issued. There are two loads executing at cycle *n* + 2 and two more loads waiting to execute in the reservation stations. The Power3 can only execute two loads per cycle, so the add (*fa*) instruction cannot execute until the loads execute, and the store instruction (*stfd*) cannot execute until the add executes. At some point this instruction mix fills the reservation stations with instructions waiting to execute, which makes the available room in the instruction buffer vary throughout the execution of the program. Dependences among the instructions being executed can exacerbate this situation. Therefore, the size of the block of instructions fetched into the instruction buffer may not be an eight-instruction block as initially assumed but a smaller size block, causing the number of L1-Icache hits to increase.

```
lfd 0,236(31)
lfd 13,244(31)
fa 0,0,13
stfd 0,268(31)
```

Figure 6. Common Block of Instructions Executed by L1-Icache Hit Micro-benchmarks

5. Summary and Future Work

As demonstrated in this paper hardware performance monitors can provide valuable data that can be used by the application programmer to identify causes of performance degradation and, thus, ways to enhance performance. Along these lines, hardware counters can be used to understand the effect of compiler options on the performance of a program or specific regions of code. This point was demonstrated with respect to IBM's Power3 and Power4 microprocessors. Certain compiler options, that otherwise seem safe, can degrade performance because they target accuracy instead of performance. It was also illustrated that sometimes event counter data needs calibration or can be misleading. Thus, although hardware performance monitors can provide valuable data to the application programmer, the data must be understood!

This research effort will continue by focusing on other computing platforms, e.g., the Pentium III and Intel's native compilers. Our objective in this research is to provide information to application programmers that will allow them to use and understand hardware performance counters to enhance the performance of their codes.

6. 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] DeRose, L. "The Hardware Performance Monitor Toolkit," *Proceedings of Euro-Par*, August 2001, pp. 122-131.

[3] DeRose, L., Ekanadham, K., Hollingsworth, J. K., and Sbaraglia, S. "SIGMA: A Simulator Infrastructure to Guide Memory Analysis", *Proceedings of Supercomputer* 2002, Baltimore, MD.

[4] IBM. "AIX 5.2 Documentation: Performance Tools Guide and Reference," http://publib16.boulder.ibm.com/pseries/en_US/infocenter/base/aix52.htm.

[5] IBM. "RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide."

[6] IBM. "XL Fortran for AIX User's Guide," Version 7, Release 1.

[7] Korn, W., P. Teller, and G. Castillo. "Just how accurate are performance counters?," *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, April 2001.

[8] Maxwell, M., P. Teller, L. Salayandia, and S. Moore. "Accuracy of Performance Monitoring Hardware," *Proceedings of the 2002 Los Alamos Computer Science Institute Symposium*, October 2002.

[9] Maxwell, M., S. Moore, and P. Teller. "Efficiency and Accuracy Issues for Sampling vs. Counting Modes of Performance Monitoring Hardware," *Proceedings of the DoD High Performance Computing Modernization Program's User Group Conference*, June 2002.

[10] Moore, S., D. Terpstra, K. London, P. Mucci, P. Teller, L. Salayandia, A. Bayona, and M. Nieto. "PAPI Deployment, Evaluation, and Extensions," *Proceedings of Users Group Conference 2003*, Bellevue, WA, June 2003.

[11] ORNL, "The Center for Computational Sciences," http://www.ccs.ornl.gov/Cheetah.html.

[12] Sadourny, R. "The Dynamics of Finite-Difference Models of the Shallow-Water Equation," *Journal of Atmospheric Sciences*, 32(4), April 1975.

[13] Salayandia, L. *A Study of the Validity and Utility of PAPI Performance Counter Data*, Master's Thesis, Department of Computer Science, University of Texas at El Paso, December 2002.

[14] SPEC,"171.spec" http://www.specbench.org/osg/cpu2000/CFP2000/171.swim/docs/171.swim.html