The Hardware Performance Monitor Toolkit

Luiz A. DeRose

Advanced Computing Technology Center, IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
laderose@us.ibm.com

Abstract. In this paper we present the Hardware Performance Monitor (HPM) Toolkit, a language independent performance analysis and visualization system developed for performance measurements of applications running on the IBM Power 3 with AIX and on Intel clusters with Linux. The HPM Toolkit supports analysis of applications written in Fortran, C, and C++. It was designed to collect hardware events with low overhead and minimum measurement error, and to display a rich set of metrics, including hints to help users in optimizing applications, without requiring modifications in the software infrastructure.

1 Introduction

Application developers have been facing new and more complex performance tuning and optimization problems as parallel architectures become more complex, with clustered SMPs, deep-memory hierarchies managed by distributed cache coherence protocols, and more intricate distributed interconnects. The sensitivity of parallel system performance to slight changes in application code, together with the large number of potential application performance problems (e.g., load balance, false sharing, and data locality) and continually evolving system software, make application tuning complex and often counter-intuitive. Thus, it is not surprising that users of parallel systems often complain that it is difficult to achieve a high fraction of the theoretical peak performance.

Correlating parallel source code with dynamic performance data from both software and hardware measurements, while still providing a portable, intuitive, and easy to use interface, is a challenging task [6]. In order to understand the execution behavior of application code in such complex environments, users need performance tools that are able to access hardware performance counters and map the resulting data to source code constructs. Moreover, these tools should be able to help the user to identify the causes of the performance problems, and not only display raw values for the hardware metrics. Without such tools, the optimal use of high-performance parallel systems will remain limited to a small group of application developers willing to master the intricate details of processor architecture, system software, and compilation systems.

To provide a system for performance measurements and visualization of applications, we developed the *Hardware Performance Monitor (HPM) Toolkit*, which is currently composed of three modules: an utility to start an application, providing performance data at the end of execution, an instrumentation library with multi-language support, and a graphical user interface for performance visualization. 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 applications, message passing, or both paradigms.

During the execution of the instrumented code, the HPM library captures hardware performance data from each instrumented section of the program on each thread of execution. At the end of the execution it combines the collected information to compute derived metrics, such as cache hit ratio and MFLOPS rates, generating one *performance file* for each task. To allow multi language cross-architecture support, as well as, flexibility in metrics selection, the performance file is represented with a self-defined format in XML.

The main contributions of the HPM Toolkit described in this paper are: First, the exploitation of the hardware performance counters to compute and present a rich set of derived metrics. These derived metrics allow users to correlate the behavior of the application to one or more of the components of the hardware. Second, an analysis of the measurement error and a technique to reduce this error. Third, an approach to analyze the derived metrics and provide hints to help users to identify the causes of performance problems, and finally, a flexible interface, defined in XML, that is able to separate performance data presentation from language and architecture peculiarities, allowing multi-language support and architecture independence.

The remainder of this paper is organized as follows: §2 describes the components of the HPM Toolkit used to collect application performance data. §3 discuss the hardware performance counters support. §4 describes the HPM Toolkit performance visualization interface and the XML interface used to allow flexibility in metric selection. Finally, §5 presents our conclusions.

2 The HPM Data Collection System

Unfortunately, most users do not have the time or desire to learn how to use complex tools. Hence, one of the main design goals of the HPM Toolkit was to create an easy to use environment for performance analysis. The first component of the HPM Toolkit is the *hpmcount* utility. It allows users to start serial or parallel applications, and at the end of execution, it provides a summary output with wall clock time (WCT), hardware performance counters information, derived hardware metrics, and resource utilization statistics.

The hpmcount utility provides a general view of the performance of an application. However, in general, this is not enough for a more complete understanding of the application behavior. Thus, the HPM Toolkit also provides an instrumentation library, so users can place instrumentation calls in selected program regions to measured the performance of the program at a finer granularity.

The HPM library supports multiple instrumentation sections, nested instrumentation, and multiple calls to the same instrumented section. During program execution, the HPM library accumulates counts and durations for all instrumented sections of the program. When nested instrumentation is used, exclusive duration is generated for the outer sections. Average and standard deviation are provided when an instrumented section is activated multiple times. As we shall see in Section 4, performance metrics are shown for each instrumented section, allowing users to instrument an application, examine the correlation of performance metrics and source code, and use the graphical interface to re-instrument the application with the knowledge obtained.

Currently, the HPM instrumentation is inserted statically by the user. A new HPM Toolkit component under development is an utility for dynamic instrumentation of programs. This utility uses the Dynamic Probe Class Library [2] (DPCL), a layer built on top of the Dyninst API [1], to support dynamic instrumentation on multiple nodes.

One important aspect of the HPM Toolkit is the support for multi-threaded applications. Due to the current trend in computer architecture, parallel systems are being built as clusters of shared memory processor. Thus, support for the shared-memory programming paradigm (e.g., pThreads or OpenMP) is a necessary requirement for any new performance tool. In order to provide multi-thread support, the HPM library has two pairs of functions to indicate the start and end of each code region to be instrumented. One pair of functions is used for instrumentation within a parallel region on a multi-threaded application, while the other pair is used outside of parallel regions. The main difference between these functions is that the former only counts the activity of the calling thread, while the latter counts the activity of a process and all of its children.

3 Hardware Performance Counters

Although software instrumentation can capture the interaction of compiler-synthesized code with runtime libraries and system software, understanding the effects of deep-memory hierarchies, cache coherence protocols, and branch prediction requires concurrent capture of both software and hardware performance metrics. Fortunately, new microprocessors provide a set of registers for access to hardware performance data.

Hardware performance counters are special purpose registers that keep track of programmable hardware events. Since they are provided at hardware level, their main strengths are low intrusion cost, accuracy, and low overhead. On the other hand, they are still a limited resource, with current processors having between 2 and 8 counters. Moreover, hardware counters tend to be specific to each processor, are generally 32 bit, and are normally programmed at kernel level, which make them prone to frequent overflows and difficult to program.

To address these problems, vendors normally provide a software API for access to the hardware counters. On IBM systems, we use the system and kernel thread performance monitor API (PMAPI), which takes care of most of the lower level issues related to accessing hardware counters, such as handling of overflows, context switches, and thread level support. However, since a particular vendor

API does not provide portability across processors from different vendors, the HPM Toolkit was also implemented on top of PAPI [3], a system-independent interface for hardware performance counters. Using the PAPI interface, the HPM Toolkit was ported to Intel platforms under Linux, and can be easily extended to any other system supported by PAPI. This paper, however, concentrates on issues related to IBM Power 3 systems. In the reminder of this section we discuss the derived metrics supported by the HPM Toolkit, and present an analysis of overhead and measurement error.

3.1 Derived Hardware Metrics

Another weakness of hardware performance counters is that they provide only raw counts, which does not necessarily help users to identify which events are responsible for bottlenecks in the program performance. For example, the information that an executed program had x million cache misses tend to be useless for a user, unless he or she can correlate this number with other data, such as number of loads and stores. To address this problem, the HPM Toolkit calculates a rich set of derived metrics that combine hardware events and time information to provide more meaningful information, such as cache miss rates, branches miss-predicted percentage, MIPS, and MFLOPS rates. Fortunately, the IBM Power 3 processor has 8 counters, which provides enough information for the generation of several derived metrics on each program execution. On systems that have less hardware counters available, multiplexing [5] could be used under PAPI, increasing coverage, but reducing accuracy.

The HPM library allows users to specify via an input file the desired set of hardware events to be used. To facilitate use, the HPM Toolkit also provides sets of pre-defined events that can be selected via environment variables. Independently of the mechanism used to select the hardware events, the library identifies the events being used and generates all possible derived metrics for the events selected. A list of the current set of derived metrics supported for the IBM Power 3 is presented in Table 1. Each of these metrics allows users to correlate the behavior of the application to one or more of the components of the hardware.

3.2 Instrumentation Overhead and Measurement Errors

As mentioned above, some of the strengths of hardware performance counters are low overhead and accuracy. Thus, it is important that performance tools based on hardware performance counters preserve these features. Two issues are considered here: *instrumentation overhead* and *measurement error*. Any software instrumentation is expected to incur in some overhead [4]. Thus, since it is not possible to eliminate the overhead, our goal was to minimize it. On the HPM library, the observed overhead for each instrumented code section, generated by the start and stop of data collection calls, is in the order of 2500 cycles (about 6.7 μsec on a 375 MHz processor). During this time, the library executes about 3000 instructions. The bulk of these operations are fixed point (about 2100),

Table 1. Supported derived metrics on the IBM Power 3

Derived Metric	Method
Total time in user mode	Cycles/CPU frequence
Utilization rate	Total time in user mode/WCT
IPC	Instructions completed/Cycles
MIPS	Instructions completed/(1000000 × WCT)
Instructions per IC Miss	Instructions completed/Instructions cache misses
Total LS operations	Loads + Stores
% of cycles LSU is idle	100 × LSU idle/Cycles
Instructions per LS	Instructions completed/Total LS
Loads per load miss	Loads/L1 Load misses
Stores per store miss	Stores/L1 Store misses
Loads per L2 load miss	Loads/L2 Load misses
Stores per L2 store miss	Stores/L2 Store misses
Loads per TLB miss	Loads/TLB misses
Load stores per D1 miss	Total LS/(L1 Load misses + Store misses)
L1 cache hit rate	100 × (1 − ((L1 Load misses + Store misses)/Total LS)
L2 cache hit rate	100 × (1 − ((L2 Load misses + Store misses)/Total LS)
Snoop hit ratio	100 × Snoop hits/Snoop requests
HW FP instructions per cycle	$(FPU_0 + FPU_1)$/Cycles
Float point operations	$FPU_0 + FPU_1$ + FMAs
Float point operations rate	$(FPU_0 + FPU_1 + \text{FMAs})/(1000000 \times \text{WCT})$
Computation intensity	$(FPU_0 + FPU_1 + \text{FMAs})$/Total LS
FMA %	$100 \times (\text{FMAs executed} \times 2)/(FPU_0 + FPU_1 + \text{FMAs})$
Fixed point instructions	$FXU_0 + FXU_1 + FXU_2$
Branches Misspredicted %	100 × Branches Misspredicted/Branches

loads (445), and stores (325). Most of the overhead is due to time measurement, since for portability we use the function gettimeofday, that unfortunately tends to be an expensive operation in most systems. Since we are accessing the hardware counters, we could have avoided this overhead by deriving time using cycles and the frequency rate of the processor. However, this approach would reduce the number of available counters for user selection, which could be a problem, specially when only a few hardware counters are available. Moreover, since the PMAPI takes care of context switches and thread accumulation, this approach would measure the total time in user mode, not wall clock time, which is a more interesting metric for program optimization. Therefore, we decided not to use this approach.

Several issues were considered in order to reduce measurement error. First, most of the library operations are executed before starting the counters, when returning the control to the program, or after stopping the counters, when the program calls a "stop" function. However, even at the library level, there are a few operations that must be executed within the counting process, as for ex-

Table 2. Counter measurements for timing and counting functions within HPM

Function	Cycles	Instr.	Stores	Loads	FX Ops
Count	285	234	40	69	93
Time	2000	2200	48	53	2048

Table 3. Average metric values and standard deviation for the start and stop functions

	Cycles	IC Miss	LD Miss	TLB Miss
Average	285	0.350	0.004	0.023
Standard deviation	15.27	1.216	0.077	0.150

ample, releasing a lock. Second, since timing collection and capture of hardware counters information are two distinctive operations, we had to decide between timing the counters or counting the timer. We used the hardware counters to measure the overhead of both operations (i.e., start and stop the counters, and calling the timing function twice, which would correspond to the timing calls at the beginning and end of one instrumented section). As shown in Table 2, timing is about one order of magnitude more expensive than counting. Thus, the counters calls are wrapped by the timer calls, generating a small error in the time measurement (in the order of 0.8 μsec on a 375 MHz processor), but minimal error in the counting process.

However, in order to access and read the counters, the library still has to call lower level routines from the PMAPI. Although one of the last instructions executed by the library before returning the control to the program is a PMAPI call to start the counters, and the first instruction executed by a "stop" function is a call to stop the counters, there are always some instructions executed by the kernel that are accounted as part of the program. Also, cache or TLB misses occurred while executing library calls can generate measurement errors. So, in order to compensate for these measurement errors, we use the hardware counters to measure the cost of one call to the start and stop functions. This measurement is performed twice, during initialization and finalization of the library, and we consider the minimum of these calls as measurement error, and subtract these values from the values obtained on each instrumented code section.

In order to estimate the final measurement error, we called the start and stop functions 1000 times, counting all pre-defined set of events provided in the library. We observed that the number of completed instructions, loads, and stores were constant (the values shown on Table 2), while there was a small variation on cycles and number of misses (TLB, instruction, and loads[1]), as shown in Table 3. The standard deviations presented in Table 3 correspond to the measurement error for each of the main hardware events measured by the library.

[1] The number of store misses was always zero.

4 Performance Visualization

One of the design goals of the HPM Toolkit was to create an easy to use cross-architecture, language independent performance analysis interface. Hence, the implementation of the performance visualization component (*hpmviz*) relies on a single interface for performance visualization that provides a source code editor. This editor allows users to refine the performance analysis by re-instrumenting the application, while visualizing performance data from earlier executions. Additionally, one can access and load performance data from multiple executions, including different numbers of processors and different hardware count events. This functionality allows users to compare executions, in order to better understanding hardware and software interactions. In this section we present the main functionality of hpmviz, and a description of the XML interface used as input.

4.1 Hpmviz

Hpmviz takes as input the performance files generated by the HPM library. Users can visualize a single performance file from a parallel execution or multiple files from the same execution. In the latter case, hpmviz takes care of merging the files. As Shown in Figure 1, which displays an instrumented version of a mixed mode implementation of the swim code from the SPEC CPU benchmark, the main window of hpmviz is divided in two panes. The left pane displays for each instrumented section an identification, inclusive duration, exclusive duration, and count[2]. The instrumented sections are sorted by exclusive duration, so users can quickly identify the major time consuming portions of the application.

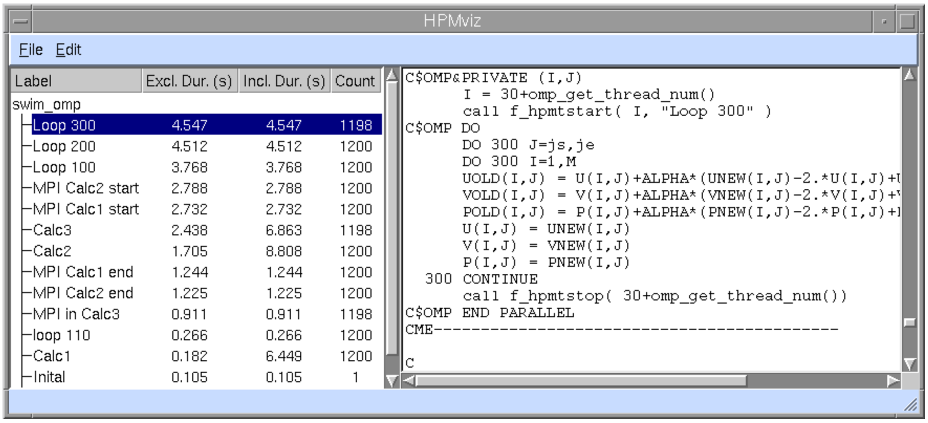
Left clicking on an instrumentation point in the left pane refocus the corresponding section in the source code pane. Right clicking on any instrumentation section in the left pane, brings a "metrics" window, shown in Figure 2, which displays all metrics for the corresponding instrumented section.

4.2 Identifying Possible Performance Problems

In order to help users identify performance problems based on the values of the derived metrics, the HPM Toolkit uses heuristics based on the characteristics of the architecture, to define a range of values considered satisfactory for some of the metrics (the metrics in bold in Table 1). When a metric value is below the threshold predefined as minimum recommended value for the metric, it appears in the metrics window highlighted with red. Similarly, when the metric value is above the predefined threshold value, it appears highlighted with green.

These threshold values were defined and fine-tuned based on the understanding of the architecture and feedback from application experts. A green value indicates that the hardware components addressed by the metric are being well utilized, while a red value indicates that the corresponding section of the code may

[2] These values correspond to the maximum value of the metric, across the parallel execution of the program.

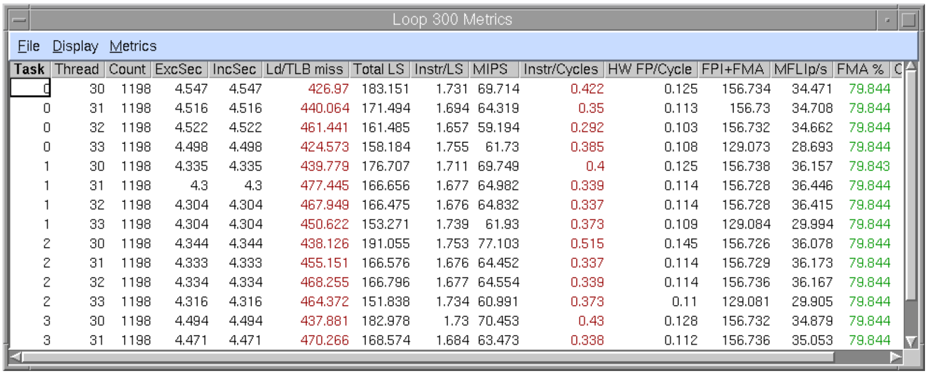


Label	Excl. Dur. (s)	Incl. Dur. (s)	Count
swim_omp			
Loop 300	4.547	4.547	1198
Loop 200	4.512	4.512	1200
Loop 100	3.768	3.768	1200
MPI Calc2 start	2.788	2.788	1200
MPI Calc1 start	2.732	2.732	1200
Calc3	2.438	6.863	1198
Calc2	1.705	8.808	1200
MPI Calc1 end	1.244	1.244	1200
MPI Calc2 end	1.225	1.225	1200
MPI in Calc3	0.911	0.911	1198
loop 110	0.266	0.266	1200
Calc1	0.182	6.449	1200
Inital	0.105	0.105	1

```
C$OMP&PRIVATE (I,J)
      I = 30+omp_get_thread_num()
      call f_hpmtstart( I, "Loop 300" )
C$OMP DO
      DO 300 J=js,je
      DO 300 I=1,M
      UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+U
      VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+V
      POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+P
      U(I,J) = UNEW(I,J)
      V(I,J) = VNEW(I,J)
      P(I,J) = PNEW(I,J)
  300 CONTINUE
      call f_hpmtstop( 30+omp_get_thread_num())
C$OMP END PARALLEL
CME------------------------------------------

C
```

Fig. 1. Hpmviz main window



Task	Thread	Count	ExcSec	IncSec	Ld/TLB miss	Total LS	Instr/LS	MIPS	Instr/Cycles	HW FP/Cycle	FPI+FMA	MFLIp/s	FMA %	C
0	30	1198	4.547	4.547	426.97	183.151	1.731	69.714	0.422	0.125	156.734	34.471	79.844	
0	31	1198	4.516	4.516	440.064	171.494	1.694	64.319	0.35	0.113	156.73	34.708	79.844	
0	32	1198	4.522	4.522	461.441	161.485	1.657	59.194	0.292	0.103	156.732	34.662	79.844	
0	33	1198	4.498	4.498	424.573	158.184	1.755	61.73	0.385	0.108	129.073	28.693	79.844	
1	30	1198	4.335	4.335	439.779	176.707	1.711	69.749	0.4	0.125	156.738	36.157	79.843	
1	31	1198	4.3	4.3	477.445	166.656	1.677	64.982	0.339	0.114	156.728	36.446	79.844	
1	32	1198	4.304	4.304	467.949	166.475	1.676	64.832	0.337	0.114	156.728	36.415	79.844	
1	33	1198	4.304	4.304	450.622	153.271	1.739	61.93	0.373	0.109	129.084	29.994	79.844	
2	30	1198	4.344	4.344	438.126	191.055	1.753	77.103	0.515	0.145	156.726	36.078	79.844	
2	31	1198	4.333	4.333	455.151	166.576	1.676	64.452	0.337	0.114	156.729	36.173	79.844	
2	32	1198	4.334	4.334	468.255	166.796	1.677	64.554	0.339	0.114	156.736	36.167	79.844	
2	33	1198	4.316	4.316	464.372	151.838	1.734	60.991	0.373	0.11	129.081	29.905	79.844	
3	30	1198	4.494	4.494	437.881	182.978	1.73	70.453	0.43	0.128	156.732	34.879	79.844	
3	31	1198	4.471	4.471	470.266	168.574	1.684	63.473	0.338	0.112	156.736	35.053	79.844	

Fig. 2. Metrics window

need optimization with regards to the utilization of the hardware components covered by the metric. Consider for example the derived metric computation intensity, which is the ratio of number of floating-point operations by the number of array loads and stores. The Power 3 architecture has two floating-point units (FPU) and two load-store units (LSU). Each FPU can execute a multiply, an add, or a fused-multiply-add (FMA) per cycle. Therefore, computation intensity of 2 would represent the best utilization of these hardware components (FPU and LSU) when performing floating-point operations. Similarly, to achieve the best possible utilization of the FPU, the program should have FMA percentage close to 100%. However, since the threshold values depend on the context of the program, the red and green colors are only considered as hints to the user, and not as hard indication of a performance problem. For instance, if the measured section of the program, in the example above, was performing mostly communications or fixed-point operations, The tool would probably indicate a poor FPU utilization, which is true, but would not be necessarily the reason for a performance problem.

4.3 Performance Files

An important aspect of the HPM performance visualization system is its multi-language support and architecture independence. The problem in providing such functionality is that the performance visualization system should be built such that it can work with minimum knowledge of the data that is going to be displayed. In order to be able to provide such functionality, it was necessary to define a flexible interface that is able to separate performance data presentation from language and architecture peculiarities. Only with this separation it is possible to display any kind of metrics, as well as, add new metrics and support new languages and architectures, without requiring extensive modifications to the graphical user interface. The HPM Toolkit performance file was designed to provide the generality and extensibility necessary to represent a diverse set of performance metrics. XML was chosen as the language for the data format because it provides this flexibility. In addition, by using XML, we could use its parsing framework, which allowed a quicker development, since very little new code had to be built over the framework.

The performance file consists of one set of data for each instrumented record in the code. Due to the support of threaded and MPI programs, an instrumented section of the program may generate several instrumentation records. To be able to present the performance data grouped by instrumented sections of the program, and not by records, we defined a unique identification for each instrumented section based on the file name and the line numbers. This information is transparently inserted into the library calls during a pre-processor phase at compile time. Each of these performance records contains specifications that include both mandatory and optional data fields, as shown in Figure 3, which displays the XML definition for one instrumentation record.

The combination of metadata and data defined for the performance file is the key to the HPM Toolkit extensibility. It allows hpmviz to render the necessary information for presentation, such as the metrics being displayed, as well as, presentation characteristics, like the range of pre-defined thresholds and display defaults (e.g., if the metric should be displayed or not).

5 Conclusions

In this paper, we described the HPM Toolkit, a language independent performance analysis and visualization system developed for performance measurements of sequential and parallel applications running on the IBM Power 3 and on Intel clusters with Linux.

The HPM Toolkit supports instrumentation and analysis of MPI and multi-threaded applications written in Fortran, C, and C++. One of its main strengths

```
<InstrumentationPt lid="swim.f775" disp="true" tid="37" task="0"
       label="Loop 300" file="swim.f" linestart="775" lineend="786" >
  <data name="Count" value="1198" />
  <data name="ExcSec" value=" 2.918" />
  <data name="IncSec" value=" 2.918" />
  <data name="PM_CYC" value=" 169719339" disp="false" />
  <data name="PM_INST_CMPL" value=" 135603862" disp="false" />
  <data name="PM_ST_CMPL" value=" 8255773" disp="false" />
  <data name="PM_LD_CMPL" value=" 56883532" disp="false" />
  <data name="PM_FPU0_CMPL" value=" 6463248" disp="false" />
  <data name="PM_FPU1_CMPL" value=" 5122784" disp="false" />
  <data name="PM_EXEC_FMA" value=" 7697521" disp="false" />
  <data name="User time" value=" 0.764" />
  <data name="Use rate" value=" 0.262" />
  <data name="Total LS" value=" 65.139" />
  <data name="Instr/LS" value=" 2.082" sbad="-1.0" sgood="2.5" dbad="-1.0" dgood="2.5" />
  <data name="MIPS" value=" 46.476" />
  <data name="IpC" value=" 0.799" sbad="-1.0" sgood="2.0" dbad="-1.0" dgood="2.0" />
  <data name="HW FP/Cyc" value=" 0.068" />
  <data name="FPI+FMA" value=" 19.284" />
  <data name="Mflip/s" value=" 6.609" />
  <data name="FMA %" value=" 79.835" sbad="-40.0" sgood="70.0" dbad="-40.0" dgood="70.0" />
  <data name="Comp Int." value=" 0.296" sbad="-0.7" sgood="1.4" dbad="-0.7" dgood="1.4" />
</InstrumentationPt>
```

Fig. 3. XML definition for one instrumentation record

is the collection of hardware events with low overhead and minimum measurement error, for the presentation of a rich set of derived metrics, including hints to help users in optimizing applications.

In order to provide multi-language support and architecture independence, we defined a flexible and extensible format for the performance file, which allows measurement of different metrics, without requiring major modifications in the software infrastructure.

References

1. Buck, B. R., and Hollingsworth, J. K. An API for Runtime Code Patching. *Journal of High Performance Computing Applications 14*, 4 (Winter 2000). 124
2. DeRose, L., Hoover Jr., T., and Hollingsworth, J. K. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium* (April 2001). 124
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14:3, Fall 2000. 125
4. Malony, A. D., Reed, D. A., and Wijshoff, H. A. G. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems 3*, 4 (July 1992), pp. 433–450. 125
5. May, J. M. MPX: Software for multiplexing hardware performance counters in multithreaded programs. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium* (April 2001). 125

6. Pancake, C. M., Simmons, M. L., and Yan, J. C. Performance Evaluation Tools for Parallel and Distributed Systems. *IEEE Computer 28*, 11 (November 1995). 122