A Hardware Counters Based Tool for System Monitoring

Tiago C. Ferreto[1]*, Luiz DeRose[2], and César A.F. De Rose[1]

[1] Catholic University of Rio Grande do Sul (PUCRS),
Post-Graduate Program on Computer Science, Porto Alegre, Brazil
{tferreto,derose}@inf.pucrs.br
[2] IBM T.J. Watson Research Center,
Yorktown Heights, NY, USA
laderose@us.ibm.com

Abstract. In this paper we describe the extensions to the RVision tool to support hardware performance counters monitoring at system level. This monitoring tool is useful for system administrators to detect applications that need tuning. We present a case study using a parallel version of the Swim benchmark from the SPEC suite, running on an Intel Pentium III Linux cluster, where we show a performance improvement of 25%. In addition we present some intrusion measurements showing that our implementation has very low intrusion even with high monitoring frequencies.

1 Introduction

As parallel architectures become more complex it is becoming much more difficult for applications to run at a reasonable fraction of the peak performance of parallel systems. In order to improve program performance, experienced program developers have been using hardware performance counters for application tuning [1]. Likewise, in order to help programmers tune their applications, a variety of utilities and libraries have been created to provide access to the hardware performance counters at user level [2,3,4,5,6, 7,8].

Unfortunately, for most users, it is not always clear when their programs need tuning. Thus, system administrators would like to be able to monitor application performance metrics, such as MFlops/sec and cache hit ratios, to identify programs that could be candidates for optimization. However, there are practically no system-monitoring tools available on cache-based systems that provide such information. An approach being used today by system administrators is to apply "*wrappers*" to job submission scripts that activate utilities to collect hardware performance information at the application level, in order to generate summary files at the end of the execution. An example of such an approach is the "*hpmcollect*" interface [9], developed at the Scientific Supercomputing Center at the University of Karlsruhe, which automatically starts an utility [6] to collect hardware performance counters data for all applications submitted for execution, and at the end of the execution, collects and combines the hardware counters output from all parallel tasks, providing a short overview of the total performance and the resource usage of the parallel application.

* Work supported by HP-Brazil

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 7–16, 2003.

In order to provide a more complete solution for system administrators, we extended the RVision tool for cluster monitoring [10] with a monitoring library to collect hardware performance counters information during program execution, and a monitoring client for presentation in the forms of graphs and tables of the hardware events and derived metrics. In this paper we describe the implementation and the main features of this monitoring system, which include the easy of use, the flexibility in selecting events to be monitored, and its very low intrusion. In addition, we demonstrate the value of this system with an example using the SPEC Swim benchmark [11].

The remainder of this paper is organized as follows. We begin in Section 2 describing the utilization of hardware counters for program optimization and explaining the design and implementation of the modules for Hardware Performance Counters monitoring using RVision. In Section 3 we present intrusion measurements. In Section 4 we present an utilization example of the RVision hardware monitoring module. Finally, we summarize our conclusions and directions for future work in Section 5.

2 Hardware Counters Monitoring

Hardware performance counters are special purpose registers available in most modern microprocessors that keep track of programmable hardware events at every cycle. These events represent hardware activities from all functional units of the processor, allowing the low overhead access to hardware performance information, such as counts of instructions, cache misses, and branch misprediction. These registers were originally designed for hardware debugging, but, although the number of counters and type of events available differs significantly between microprocessors, most of them provide a common subset that allows programmers to use these counters to compute derived metrics to correlate the behavior of the application to one or more of the hardware components. With the availability of kernel level APIs to access the hardware counters at a user level, as well as performance tools and libraries that provide event counts and derived metrics, hardware counters have become an invaluable asset for application performance tuning.

In contrast, system administrators have not been able to exploit effectively the availability of hardware counters, mainly due to the lack of monitoring systems capable of accessing the counters. The access to derived hardware metrics during program execution would be helpful to system administrators to detect applications that need tuning. For example, the value of a particular metric falling constantly below a pre-defined threshold would be an indication that the program is a candidate for optimization.

For our hardware performance counters monitoring approach, we extended RVision, a multi-user monitoring system for GNU/Linux based clusters [10] developed at the Research Center in High Performance Computing (CPAD - PUCRS/HP). It requires the availability of a kernel interface to access the hardware performance counters at a system level, and we consider that programs will have dedicated use of the nodes during execution, which is normally the case on the majority of supercomputing sites.

The RVision monitor was originally developed to acquire information such as processor and memory usage, through kernel system calls. Due to RVision's open architecture, only a new *Monitoring Library* and a new *Monitoring Client*, which are described next, were needed to obtain the hardware counters values. The "Monitoring Library" is re-

sponsible for the capture of selected hardware events, while the "Monitoring Client" is responsible for presenting this new information, which also includes derived performance metrics. We implemented and tested these components on the "*Tropical*" Linux Cluster at CPAD, which has 8 dual-processor (Pentium III-1GHz) nodes, switched via a Fast-Ethernet network. Each node has 256 MBytes of main memory. Each processor has two levels of cache: 32 KBytes of Level 1 (with 4-way set associativity) and 256 KBytes of Level 2 (with 8-way set associativity). In both cases the cache line size is 32 bytes.

2.1 Monitoring Library

Currently, Linux does not provide an interface to access the hardware performance counters. Hence, we patched the Linux kernel using the "perfctr" patch and driver, developed by Mikael Petterson [2]. This patch provides a user and a system level interface to access the performance monitoring counters on Intel X86 processors. Since the number of counters and type of events available differs significantly between microprocessors, the RVision Monitoring Library to capture information provided by the hardware counters is architecture dependent. However, our monitoring library can be easily ported to any other platform that provides a kernel level application programming interface to access the hardware counters at system level.

The selection of events to be monitored were restricted to the hardware counter events available on the Pentium III processor, which provides two performance monitoring counters capable of counting a total of 77 different events (at most two at a time). In addition, the architecture provides a time stamp counter (TSC), which counts the elapsed machine cycles, as well as the CPU frequency. The complete specification of the performance monitoring counters and the description of all of its events are presented in [12].

From the events provided by the Pentium III architecture, we selected the following set of pair of events (in addition to the TSC) to be used for monitoring:

- P6_INST_RETIRED and P6_CPU_CLK_UNHALTED, to count the number of instructions completed and the number of machine cycles used by the program.
- P6_FLOPS and P6_CPU_CLK_UNHALTED, to count the number of floating point instructions and the number of machine cycles used by the program.
- P6_DATA_MEM_REFS and P6_DCU_LINES_IN, to count the number of level 1 accesses and the number of level 1 cache misses, respectively.
- P6_L2_RQSTS and P6_L2_LINES_IN, to count the number of level 2 accesses and the number of level 2 cache misses, respectively.
- P6_INST_DECODED and P6_INST_RETIRED, to count the number of instructions dispatched and the number of instructions decoded.

Depending on the set of counters used, we compute the following derived metrics:

MIPS: the average number of instructions per second (in millions), computed as:
P6_INST_RETIRED/(1000000 * TSC/CPU frequency)

Utilization Rate: the ratio of CPU time to wall clock time, computed as:
P6_CPU_CLK_UNHALTED/TSC

IpC: Instructions per Cycle, computed as:
$\text{P6\_INST\_RETIRED}/\text{P6\_CPU\_CLK\_UNHALTED}$
MFlops/sec: Millions of floating point operations per second, computed as:
$\text{P6\_FLOPS}/(1000000 * \text{TSC}/\text{CPU frequency})$
Level 1 cache hit ratio: Computed as:
$100 * (1 - (\text{P6\_DCU\_LINES\_IN}/\text{P6\_DATA\_MEM\_REFS}))$
Level 2 cache hit ratio: Computed as:
$100 * (1 - (\text{P6\_L2\_LINES\_IN}/\ \text{P6\_L2\_RQSTS}))$
Percentage of instructions dispatched that completed: Computed as:
$100 * (\text{P6\_INST\_RETIRED}/\text{P6\_INST\_DECODED})$

2.2 Monitoring Client

The monitoring client is responsible for the presentation of tables and graphics with the performance monitoring counters information and the derived metrics. The communication routines of the monitoring client were implemented in C, and the GUI was implemented in Java, with JNI being used to provide the connection between the languages. A snapshot of the monitoring client is presented in Figure 3. It provides a table presenting the hardware counters values and the derived metric values for each node of the cluster, as well as a histogram containing the average of the derived metric for all processors. The histogram scrolls to the left, presenting the most recent information at the rightmost side.

3 Intrusion Measurement

The main concern when a tool is used to monitor some resource is how much the readings are being affected by the tool. Being a parallel application itself, the monitoring tool, when active, is consuming cluster resources such as CPU and network bandwidth. This intrusion should be minimal to guarantee that the monitored data is accurate and that the behavior of the other application running on the cluster is not considerably affected.

We measured RVision's intrusion by defining a monitoring session capturing hardware counters values for all cluster nodes and requesting this data using online monitoring with a regular time interval. Different applications are executed in the cluster with and without monitoring and we compared the execution times. We varied the monitoring time interval from 2 seconds to 200 milliseconds to simulate a worst-case scenario.

To evaluate RVision under different workloads we used the following programs from the NAS Parallel Benchmarks [13,14]: Embarassingly Parallel (EP), Integer Sort (IS), LU Decomposition (LU), Conjugate Gradient (CG), and Multigrid (MG). All benchmarks were compiled using class A, defined internally in the NPB, and executed on the 8 nodes of the Tropical Linux Cluster at CPAD [15]. Table 1 presents the intrusion results for all test cases. In each case the test application was executed 10 times for each time interval with monitoring turned on.

All benchmarks used in the measurement generated low intrusivity. The worst-case scenario, using 200 milliseconds as time interval, presented intrusion values of less than 1% for all programs with the exception of the IS benchmark. The Integer Sort benchmark

Table 1. Intrusion measurements

Time Interval (sec)	EP	IS	LU	CG	MG
2.0	0.22%	0.12%	0.09%	0.15%	0.03%
1.8	0.22%	0.15%	0.17%	0.33%	0.10%
1.6	0.23%	0.72%	0.10%	0.44%	0.12%
1.4	0.39%	0.85%	0.11%	0.49%	0.12%
1.2	0.50%	0.86%	0.24%	0.53%	0.17%
1.0	0.54%	0.86%	0.34%	0.53%	0.17%
0.8	0.55%	1.21%	0.38%	0.59%	0.21%
0.6	0.56%	1.61%	0.39%	0.63%	0.29%
0.4	0.66%	1.91%	0.54%	0.64%	0.33%
0.2	0.72%	2.13%	0.71%	0.67%	0.51%

has a high network utilization. Hence, it was more affected by the traffic generated by the on-line monitoring.

4 Example of Use

In order to demonstrate the usefulness of the hardware counters monitoring feature of RVision, we ran a parallel version of the SPEC Swim benchmark with problem size defined by N1=N2=2048, on the Tropical Linux cluster, and monitored both the MFlops/sec rate and the level 1 cache hit ratio during the execution of the program, using 1 second as monitoring interval. Figure 1 presents the system's average L1 hit ratio when the program started its execution, we observe that the cache hit ratio dropped considerable (most recent information is presented at the rightmost side). Looking at Figure 2, which presents a table with a snapshot with the monitoring values for all processors, we observe that the level 1 hit ratio was in the order of 54% in all processors, indicating that the application needed some sort of program restructuring, due to its poor utilization of the memory subsystem. This poor cache utilization reflects in the MFlops/sec rate, shown in Figure 3, which indicates a sustained performance of around 67 MFlops/sec.

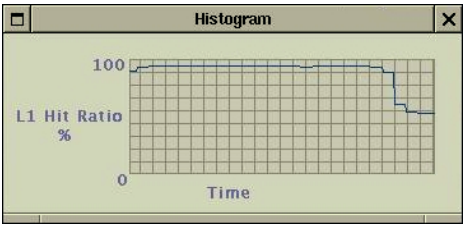
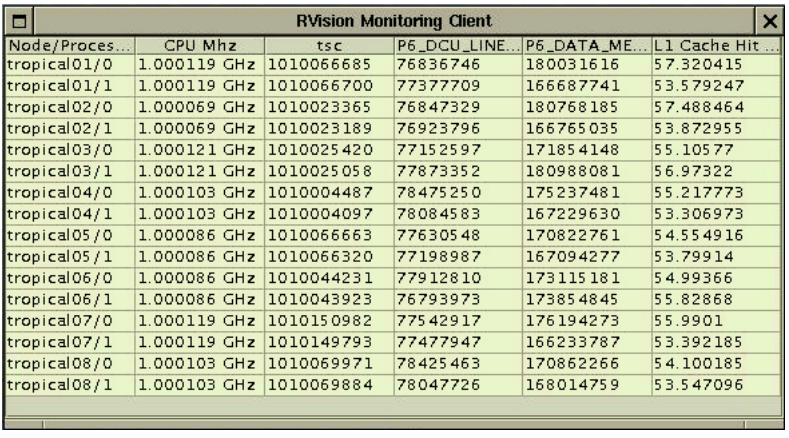


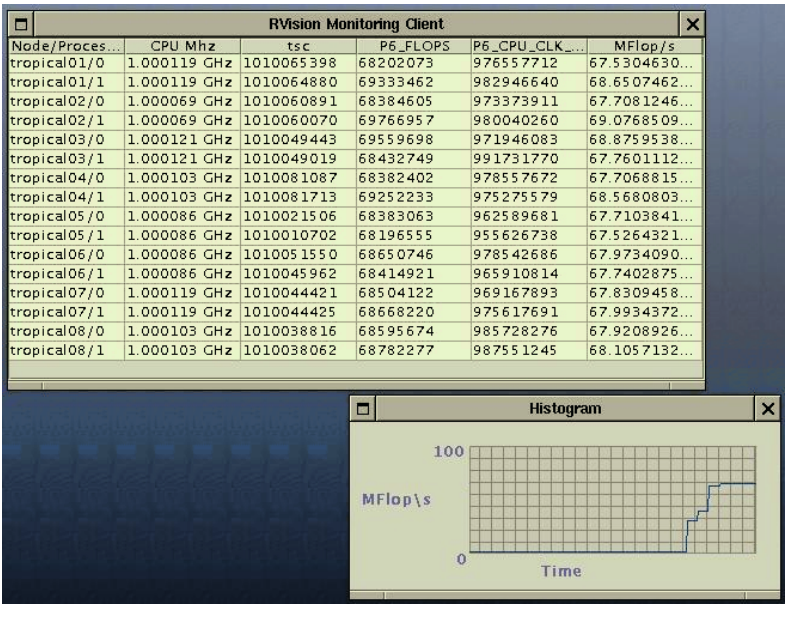
Fig. 1. System's average L1 hit ratio running the SPEC Swim benchmark



RVision Monitoring Client

Node/Proces...	CPU Mhz	tsc	P6_DCU_LINE...	P6_DATA_ME...	L1 Cache Hit ...
tropical01/0	1.000119 GHz	1010066685	76836746	180031616	57.320415
tropical01/1	1.000119 GHz	1010066700	77377709	166687741	53.579247
tropical02/0	1.000069 GHz	1010023365	76847329	180768185	57.488464
tropical02/1	1.000069 GHz	1010023189	76923796	166765035	53.872955
tropical03/0	1.000121 GHz	1010025420	77152597	171854148	55.10577
tropical03/1	1.000121 GHz	1010025058	77873352	180988081	56.97322
tropical04/0	1.000103 GHz	1010004487	78475250	175237481	55.217773
tropical04/1	1.000103 GHz	1010004097	78084583	167229630	53.306973
tropical05/0	1.000086 GHz	1010066663	77630548	170822761	54.554916
tropical05/1	1.000086 GHz	1010066320	77198987	167094277	53.79914
tropical06/0	1.000086 GHz	1010044231	77912810	173115181	54.99366
tropical06/1	1.000086 GHz	1010043923	76793973	173854845	55.82868
tropical07/0	1.000119 GHz	1010150982	77542917	176194273	55.9901
tropical07/1	1.000119 GHz	1010149793	77477947	166233787	53.392185
tropical08/0	1.000103 GHz	1010069971	78425463	170862266	54.100185
tropical08/1	1.000103 GHz	1010069884	78047726	168014759	53.547096

Fig. 2. System's L1 hit ratio running the SPEC Swim benchmark



RVision Monitoring Client

Node/Proces...	CPU Mhz	tsc	P6_FLOPS	P6_CPU_CLK_...	MFlop/s
tropical01/0	1.000119 GHz	1010065398	68202073	976557712	67.5304630...
tropical01/1	1.000119 GHz	1010064880	69333462	982946640	68.6507462...
tropical02/0	1.000069 GHz	1010060891	68384605	973373911	67.7081246...
tropical02/1	1.000069 GHz	1010060070	69766957	980040260	69.0768509...
tropical03/0	1.000121 GHz	1010049443	69559698	971946083	68.8759538...
tropical03/1	1.000121 GHz	1010049019	68432749	991731770	67.7601112...
tropical04/0	1.000103 GHz	1010081087	68382402	978557672	67.7068815...
tropical04/1	1.000103 GHz	1010081713	69252233	975275579	68.5680803...
tropical05/0	1.000086 GHz	1010021506	68383063	962589681	67.7103841...
tropical05/1	1.000086 GHz	1010010702	68196555	955626738	67.5264321...
tropical06/0	1.000086 GHz	1010051550	68650746	978542686	67.9734090...
tropical06/1	1.000086 GHz	1010045962	68414921	965910814	67.7402875...
tropical07/0	1.000119 GHz	1010044421	68504122	969167893	67.8309458...
tropical07/1	1.000119 GHz	1010044425	68668220	975617691	67.9934372...
tropical08/0	1.000103 GHz	1010038816	68595674	985728276	67.9208926...
tropical08/1	1.000103 GHz	1010038062	68782277	987551245	68.1057132...

Histogram

Fig. 3. System's MFlops/sec rate running the SPEC Swim benchmark

Looking at the source code, we observe that the basic data structure of the application is defined in the "Common block" shown in Figure 4, and used in loops such as the one presented in Figure 5. The use of high powers of 2 as array dimensions (2048 in this case) would result in an excessive cache miss ratio for this application, due to the low associativity of the level 1 cache (4-way). This problem occurs because 9 different arrays

are being used in the loop, and since the size of each array is multiple of the cache size, array elements with the same indices will map to the same cache set. Since each set can accommodate only 4 different entries, this loop generates an excessive number of conflict misses. This could be a well-known fact for application tuning specialists and experienced programmers, but not necessarily well understood by scientists with no background in computer architecture. Hence, the availability of monitoring tools that could indicate to system administrators when an application may need tuning is an invaluable asset.

```
      PARAMETER (N1=2048, N2=2048)
      COMMON U(N1,N2), V(N1,N2), P(N1,N2),
     *   UNEW(N1,N2), VNEW(N1,N2), PNEW(N1,N2), UOLD(N1,N2),
     *   VOLD(N1,N2), POLD(N1,N2), CU(N1,N2), CV(N1,N2),
     *   Z(N1,N2), H(N1,N2), PSI(N1,N2)
```

Fig. 4. Definition of the main data structure in the SPEC Swim benchmark

```
DO 300 J=js,je
  DO 300 I=1,M
    UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+UOLD(I,J))
    VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+VOLD(I,J))
    POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+POLD(I,J))
    U(I,J) = UNEW(I,J)
    V(I,J) = VNEW(I,J)
    P(I,J) = PNEW(I,J)
300 CONTINUE
```

Fig. 5. Utilization of the arrays in one of loops in the SPEC Swim benchmark

In this case, the simple solution would be to "pad" the common block in Figure 4 with declaration of dummy vectors between the array declarations, as shown in Figure 6. The use of dummy vectors of 8 elements each will separate the mapping of the elements of the arrays with the same indices by one cache line, which will reduce the conflicts. When executing the modified program, we observe an increase in the Level 1 cache hit ratio to about 78%, as shown in Figure 7 and a sustained performance in the order of 84 MFlops/sec, as shown in Figure 8, which corresponds to an improvement of 25% in performance.

```
      PARAMETER (N1=2048, N2=2048)
      COMMON U(N1,N2), D1(8), V(N1,N2), D2(8), P(N1,N2),
     *   UNEW(N1,N2), D3(8), VNEW(N1,N2), D4(8), PNEW(N1,N2),
     *   D5(8), UOLD(N1,N2), D6(8), VOLD(N1,N2), D7(8),
     *   POLD(N1,N2), D8(8), CU(N1,N2), D9(8), CV(N1,N2), D10(8),
     *   Z(N1,N2), D11(8), H(N1,N2), D12(8), PSI(N1,N2)
```

Fig. 6. Padded common block for the SPEC Swim benchmark

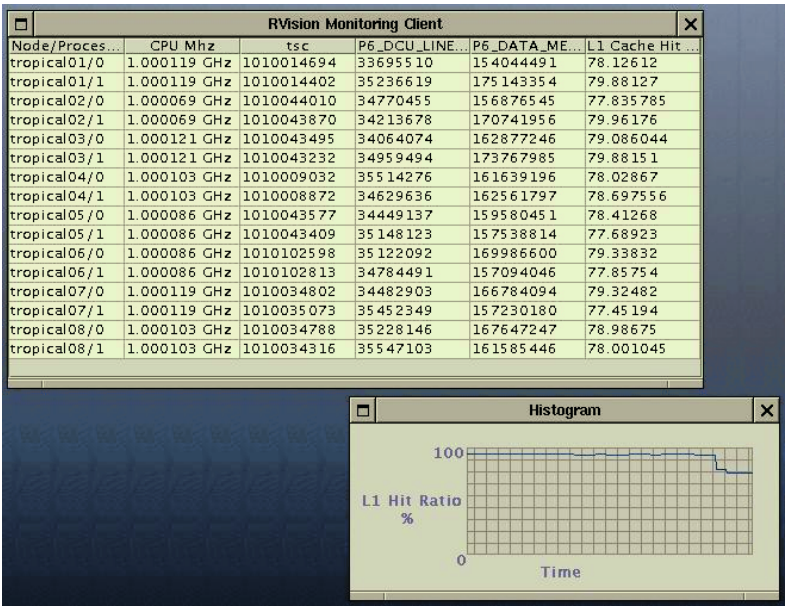


Fig. 7. System's L1 hit ratio running the padded version of the SPEC Swim benchmark

Other examples of performance problems that can be detected with this monitoring approach, which are not presented here due to space limitations are:

Utilization Rate: For a task on a dedicated compute node, this ratio should be close to 1. Lower values would indicate large system activity, which could require some application performance tuning.

IpC: Given that the processor has multiple functional units, a well tuned program should have IpC larger than 1.

Percentage of instructions dispatched that completed: This metric gives an indication of how well the speculation is working for the program. A low percentage could indicate that the program has loops with few iteration counts that could benefit from loop unrolling.

MIPS: This metric can be used for non-floating point intensive codes for monitoring of the sustained performance of the application.

L1 and L2 cache hit ratios: Are also useful to detect problems in the memory subsystem caused by capacity misses. These problems might be solved with loop transformations, such as blocking, to increase data locality.

5 Conclusions and Future Work

In this paper we presented a new approach to system monitoring for cluster architectures based on hardware counters. Our main goal is to provide an efficient tool for system

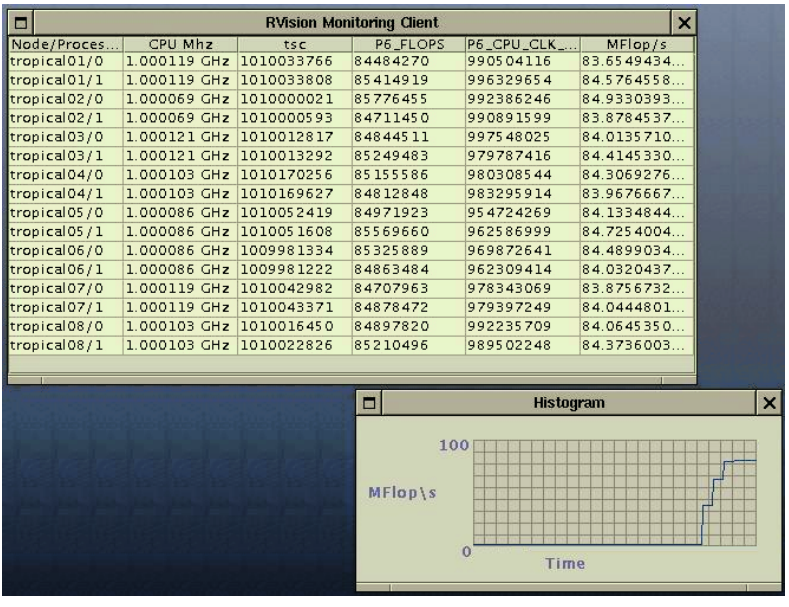


Fig. 8. System's MFlops/sec rate running the padded version of the Swim benchmark

administrators to detect programs that cause bottlenecks in cluster architectures so that throughput can be increased. As expected hardware counters monitoring has a very low intrusion resulting in more precise results. In dedicated systems it is also possible to detect applications that could need tuning, without the instrumentation needed by traditional hardware counters monitoring at application level.

To investigate this new concept we expanded our resource monitor RVision to access hardware counters on Intel Pentium III processors and to calculate derived performance metrics for Linux clusters. With this tool we analyzed the results obtained in a case study where we optimized the execution of a parallel version of the Swim benchmark from the SPEC suite, obtaining a performance increase in the order of 25%, on a Linux cluster with 16 processors. We also presented intrusion results for the expanded version of RVision running the NAS benchmark suite. We observed that in most cases intrusion is less than 1% even with high monitoring frequencies like 200 milliseconds.

We believe that systems oriented hardware monitoring tools are a very interesting alternative for system and application tuning. Our next steps is to support additional processor architectures like the Intel Pentium IV and the IBM Power4, where it will be possible to work with more derived metrics due to the increased number of hardware counters. More information about RVision and the package for download are available at *http://rvision.sourceforge.net*.

References

1. Zagha, M., Larson, B., Turner, S., Itzkowitz, M.: Performance Analysis Using the MIPS R10000 Performance Counters. In: Proceedings of Supercomputing'96. (November 1996)
2. Pettersson, M.: Linux X86 Performance-Monitoring Counters Driver. http://user.it.uu.se/~mikpe/linux/perfctr/. Computing Science Department; Uppsala University – Sweden. (2002)
3. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In: Proceedings of Supercomputing'00. (November 2000)
4. Research Centre Juelich GmbH: PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. http://www.fz-juelich.de/zam/PCL/. (2002)
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)
6. DeRose, L.: The Hardware Performance Monitor Toolkit. In: Proceedings of Euro-Par. (August 2001) 122–131
7. Janssen, C.: The visual Profiler. http://aros.ca.sandia.gov/~cljanss/perf/vprof/. Sandia National Laboratories. (2002)
8. Buck, B., Hollingsworth, J.K.: Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In: Proceedings of Supercomputing'01. (November 2001)
9. Geers, N.: Automatic Collection of HPM Data in Parallel Applications. http://www.uni-karlsruhe.de/~SP/software/tools/hpm/hpmcollect.de.html. Scientific Supercomputing Center at University of Karlsruhe. (2002)
10. Ferreto, T.C., De Rose, C.A.F., DeRose, L.: RVision: An Open and High Configurable Tool for Cluster Monitoring. In: Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid, Berlin, Germany (2002) 75–82
11. Sadourny, R.: The Dynamics of Finite-Difference Model of the Shallow-Water Equations. Journal of Atmospheric Sciences **32** (1975)
12. Intel Corporation: IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. (2002)
13. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-929, NASA Ames Research Center (1995)
14. Saphir, W., Wijngaart, R.V.D., Woo, A., Yarrow", M.: New implementations and results for the NAS parallel benchmarks 2. In: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing. (1997)
15. Research Center in High Performance Computing: `http://www.cpad.pucrs.br` (2002)