Detailed Performance Analysis Using Coarse Grain Sampling

Harald Servat, Germán Llort, Judit Giménez, and Jesús Labarta

Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
c/Jordi Girona, 31
08034 - Barcelona, Catalunya, Spain
{harald.servat,german.llort,judit.gimenez,jesus.labarta}@bsc.es

Abstract. Performance evaluation tools enable analysts to shed light on how applications behave both from a general point of view and at concrete execution points, but cannot provide detailed information beyond the monitored regions of code.

Having the ability to determine when and which data has to be collected is crucial for a successful analysis. This is particularly true for trace-based tools, which can easily incur either unmanageable large traces or information shortage.

In order to mitigate the well-known resolution *vs.* usability trade-off, we present a procedure that obtains fine grain performance information using coarse grain sampling, projecting performance metrics scattered all over the execution into thoroughly detailed representative areas.

This mechanism has been incorporated into the MPItrace tracing suite, greatly extending the amount of performance information gathered from statically instrumented points with further periodic samples collected beyond them.

We have applied this solution to the analysis of two applications to introduce a novel performance analysis methodology based on the combination of instrumentation and sampling techniques.

1 Introduction

Performance evaluation tools are able to characterize the behavior of an application using different mechanisms. However, the details obtained by these tools are strictly limited to the monitored regions of code. In order to identify the precise behavior of the application, it is required to add more and more monitors, with the consequent overhead and larger volume of data generated.

Instrumentation and sampling are two different mechanisms used to monitor applications. Instrumentation allows injection of monitors in specific points of the target application simply modifying the source code, compiling the application with special flags or specific libraries, or by using instrumentation packages like DynInst [7]. On the other hand, sampling can be used to trigger monitors at intervals or by external events and it is not related to any specific application point.

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 185–198, 2010.

We are confident that performance analysis using traces is the best approach to detail time and space variations of the behavior of applications, which can be easily masked out by profilers while summarizing information. In addition, trace visualizers provide further qualitative and quantitative analysis [17,15].

Most performance tools based on traces [22,19,1,2] rely on instrumentation to detail performance characteristics of the applications. Such tools cannot provide information beyond the instrumented points, and the granularity and size of the resulting trace strictly depends on the application structure.

Our main objective is to use basic instrumentation and coarse grain sampling so as not to increase the execution overhead and the amount of data to be analyzed, while obtaining highly detailed information. To achieve this objective we first extend the MPItrace tracing package with sampling mechanisms, and then apply a process that improves the granularity of the obtained samples, if needed.

MPItrace is a trace-based instrumentation tool that collects performance information about parallel applications. The resulting combination provides more data of the application by gathering information from uninstrumented regions of code, independent of the application structure.

The rest of this paper is structured as follows: Section 2 provides information about the implementation of the sampling mechanism combined with the MPItrace instrumentation package. Section 3 introduces the methodology used to analyze traces containing information gathered by the instrumentation and the sampling mechanisms. This methodology is evaluated in section 4. Section 5 gives an overview of related work. Finally, we plot the conclusions and future trends in Section 6.

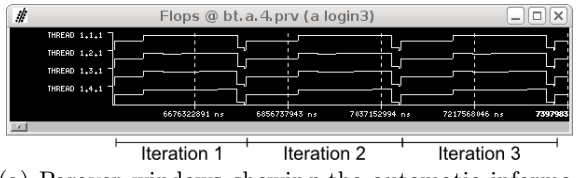
2 Instrumentation and Sampling Mechanisms

MPItrace is an instrumentation tool that collects performance information of parallel MPI applications automatically through the MPI profiling interface [11]. It also provides information of OpenMP constructs and some pthread calls by replacing the references to the runtime found in shared libraries by using dynamic library interposition mechanisms and allows the user to add his owns events in the resulting trace manually.

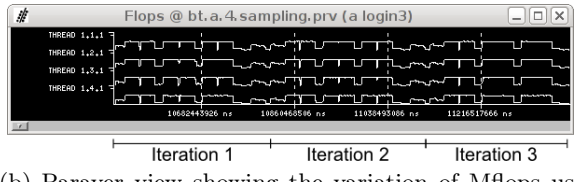
The sampling mechanism added to the MPItrace instrumentation package is built on top of PAPI [6] because it provides a flexible and precise way to determine the sampling rate [14]. This library provides a method called `PAPI_overflow` to setup the sampling frequency based on a hardware counter and a threshold. Once the selected counter reaches the given threshold, it triggers an exception that is captured by PAPI and forwarded to a callback function. The programmed callback collects processor performance metrics and the address within the process that was being executed. We consider henceforth a *sample* as the set of data containing the performance metrics on a specific time and the process address.

The utility of this framework, which extends with sampling the typical traces obtained with MPItrace, is shown in Figure 1 of the Paraver visualization tool.

Paraver represents on the y-axis the application's threads while the x-axis represents time. On both images, Paraver is configured to draw the number of floating point operations. In the left image, the number of floating points operations are drawn at instrumentation points, whereas in the right image the number of floating point operations are drawn at sampling points using a fine resolution. It is noticeable that the Figure 1(b) plots performance information not only on the default instrumented points (MPI routines in this case), but along the iteration. The figure on the right improves the quality of the analysis by detailing the achieved floating operation performance in a finer way than the figure on the left. Details that can be observed in the image on the right are: a) variance at the very beginning and end of each iteration, and, b) that every iteration has three separated regions of code that deliver a high number of floating point operations, and that each of those three regions has a short and sharp decrease of performance in the middle of the region.



(a) Paraver windows showing the automatic information gathered by the instrumentation showing the variation of Mflops



(b) Paraver view showing the variation of Mflops using a trace with data captured by instrumentation and sampling

Fig. 1. Paraver windows showing information captured by instrumentation with and without sampling mechanism

3 Analysis Methodology

The procedure we follow to characterize applications focuses on selecting the most time consuming routines, avoiding library internals and inline or intrinsic routines. We want to provide detailed performance metrics, the number of invocations, the time they need to execute and their variation across the iterations for the selected routines. This methodology requires the user to know in advance which routines are important, possibly by previously analyzing the application with profilers. We know that this is a limitation of the methodology and we

are currently working on a more automatic way to make it easier to apply. In this paper we aim to increase the details of the selected routines by using both instrumentation and sampling mechanisms.

The first step consists of obtaining a trace using the MPItrace instrumentation package with sampled information. Then the analyst will evaluate the application afterwards using Paraver. This tool provides quantitative and qualitative analysis of the traces using timelines and histograms and also contains a set of predefined windows to conduct the analysis and facilitate the characterization of the target application. Among them, there are several related windows that provide information for user routines and performance metrics.

As the knowledge about the target application may be limited, choosing a proper sampling rate to detail the application enough is a blind process. In case the selected rate results in too few metrics scattered all over the execution, they can be combined to produce highly detailed representative areas using a process that we call *folding*.

The folding process is suitable for applications based on iterative methods, which is a representative part of the applications found in high performance computing environments. For the rest of applications, a new trace could be generated with finer sampling resolution.

To proceed with the combination, the process can work at three different granularities: full iteration, instrumented user routines and running bursts (*i.e.* intervals between MPI calls).

The full iteration granularity provides a detailed view of a single iteration. Using this behavior each sample that is run within the main application loop is treated equally, independent from where it was emitted. The second granularity focuses only on user code that has been instrumented by the analyst and ignores the rest. Finally, the last option allows working separately on different intervals delimited by MPI calls. This granularity grants working only on user code and its results are not interfered by MPI calls.

We detail the folding process for full iterations in section 3.1. Applying the folding process to the rest of granularities just implies splitting the iteration every time an instrumented function or a MPI call is found.

Concerning the implementation, little modifications of the source code are required to perform the iteration folding. The user just needs to instrument the main loop so as to delimit the beginning and end of each iteration and gather the performance counter values at these points, and optionally, instrument the interesting user functions.

3.1 Folding Iterations

If the chosen sampling rate is too coarse, the performance metrics will not be precise enough to characterize any region of code. We are interested in how to obtain precise information under these circumstances. Our objective is to build a synthetic trace to provide information of the behavior of an iteration per MPI task with high precision. The subsequent analysis procedure must only target the resulting iteration and then extrapolate the results to the rest. The ability of

treating each task independently allows spotting performance differences across tasks.

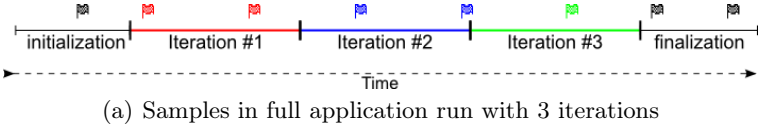
To properly obtain fine grained information relative to the performance counters we apply a two-step procedure within each task: migration of samples into a single iteration and hardware counter interpolation. To proceed with the last step, the user must indicate the desired number of output samples. Requesting more samples yields more detailed results.

Hardware Counters Folding. We define *folding* an iteration (*source*) into another (*destination*) as the operation that migrates samples from source to destination preserving the offset of source iteration and converts its performance counter values so as they are relative to the beginning of the iteration they originally belong. Figures 2(a) and 2(b) illustrate an example. The top figure shows a timeline of a program that runs 3 iterations of its main loop and the bottom figure shows the same timeline after folding the second and third iterations into the first one.

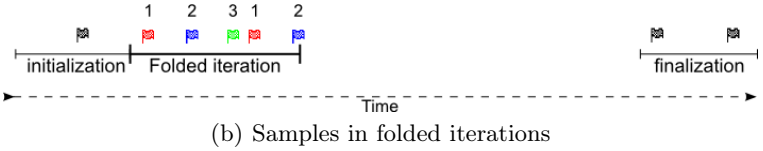
Considering applications that have a regular and iterative control and data flow and dedicated systems with low external interferences (such as preemptions, interrupts, network and memory contention, *etc...*), the resulting set of iterations conform to an ergodic system. This is to say, any iteration matches up with the typical iteration and, thus, samples can be migrated from one to another without altering the gathered values of the performance metrics.

In practice, executions are non deterministic. Even using dedicated systems, applications face different interferences each run, which may result in slight performance variations on each iteration. Considering the duration of the iteration as a normally distributed variable, we can safely remove those iterations that last more than twice the standard deviation. After this removal, we are still working with a representative set of iterations because we are keeping the iterations that their duration are within the interval of confidence of the 95% of the whole samples.

Concerning the output resolution, folding all the iterations into a single one results in a single iteration that contains all the samples scattered across



(a) Samples in full application run with 3 iterations



(b) Samples in folded iterations

Fig. 2. Flags in those figures represent points where information is located before (a) and after (b) folding all iterations. The superindices and colors of flags on Figure (b) show which was their original iteration.

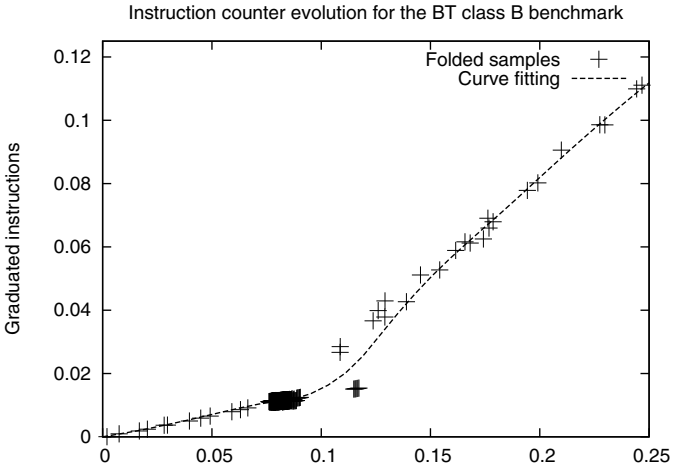


Fig. 3. Evolution of graduated instructions for the BT benchmark within an iteration

iterations, or in other words, the resolution of the folded iteration is proportional to the number of iterations folded.

Hardware Counters Interpolation. Once the hardware counter metrics have been folded into a single iteration, we take as many equidistant samples from this set of data as the user requested. Since hardware counters information is punctual in time, we build a continuous function that closely fits the set of data to estimate all the hardware counters values across the whole folded iteration.

This process is known as interpolation and in order to apply it we explored several approaches: polynomial fitting, Bézier curves [5] and Kriging [21] interpolation. Polynomial fitting requires choosing the grade of the polynomial and this cannot be done independently from the data. In addition, choosing a low order polynomial will give soft but inaccurate fitting, while a high order polynomial will fit better but will result in big fluctuations. Bézier curves do not require additional data but the points themselves. Bézier fitted well on our tests but on the stationary points. The Kriging interpolation is a general version of the Bézier curve typically used in contouring. Kriging algorithm works with the sample points plus some interpolation parameters (including fitting strictness). After some tests, we found a typical combination of parameters that fitted the samples well even in stationary points.

Although performance counters are monotonically increasing in a single iteration, it may not be completely monotonic when considering the whole set of samples. Consider the case shown in Figure 3. This figure plots the graduated instructions across time. There are several points with more graduated instructions than near points in the future due to variations of performance counters values on each iteration. Even though this effect was previously minimized by the outlier removal and by the tendency of the interpolation function to ignore

spurious values, they can still appear. Whenever this happens, they are just ignored.

This process must be repeated for each task present in the tracefile and concludes reintroducing the values of the interpolated performance counters into the trace.

4 Case Study

This section shows how the folding process and the combination of instrumented and sampled performance information contributes to improve the detailed analysis of real applications.

Table 1. Characteristics of the system used for the evaluation

Experimental systems characteristics	
Processor family	Intel Itanium 2
Processor frequency	1.6 GHz
PAPI version	3.6.2
Linux kernel	2.6.16.46-0.12
Compilers (C/Fortran)	icc and ifort 11.0

Table 2. Setup of the different experiments

Application	NAS bt.B	Alya
Number of processors used	16	4
Sampling frequency (in million cycles)	50	1000
Average duration per iteration	183ms	18.2s
Samples per iteration	5 - 6	28 - 29
Number of timesteps	200	100
Runtime overhead	1.5%	3%

To perform our analysis using the folding process we choose bt.B from the NAS MPI Parallel Benchmark Suite 3.2 [3], and Alya [12], a computational mechanics simulator that is typically run in our production environment. Table 1 describes the characteristics of the system used in this study and table 2 provides information about the setup of the different experiments. The overhead row in table 2 comprises the overhead of the sampling mechanism plus the MPI instrumentation and the manually added events used to identify the iterations.

We request a thousand samples in the target folding iteration regarding the performance metrics. This is equivalent to set a sampling frequency 200 and 35 times higher in bt.B and Alya respectively with the proportional increase of overhead. In addition to that, we use the user function granularity of the folding process in order to improve the understanding of the results by providing detailed information of representative functions.

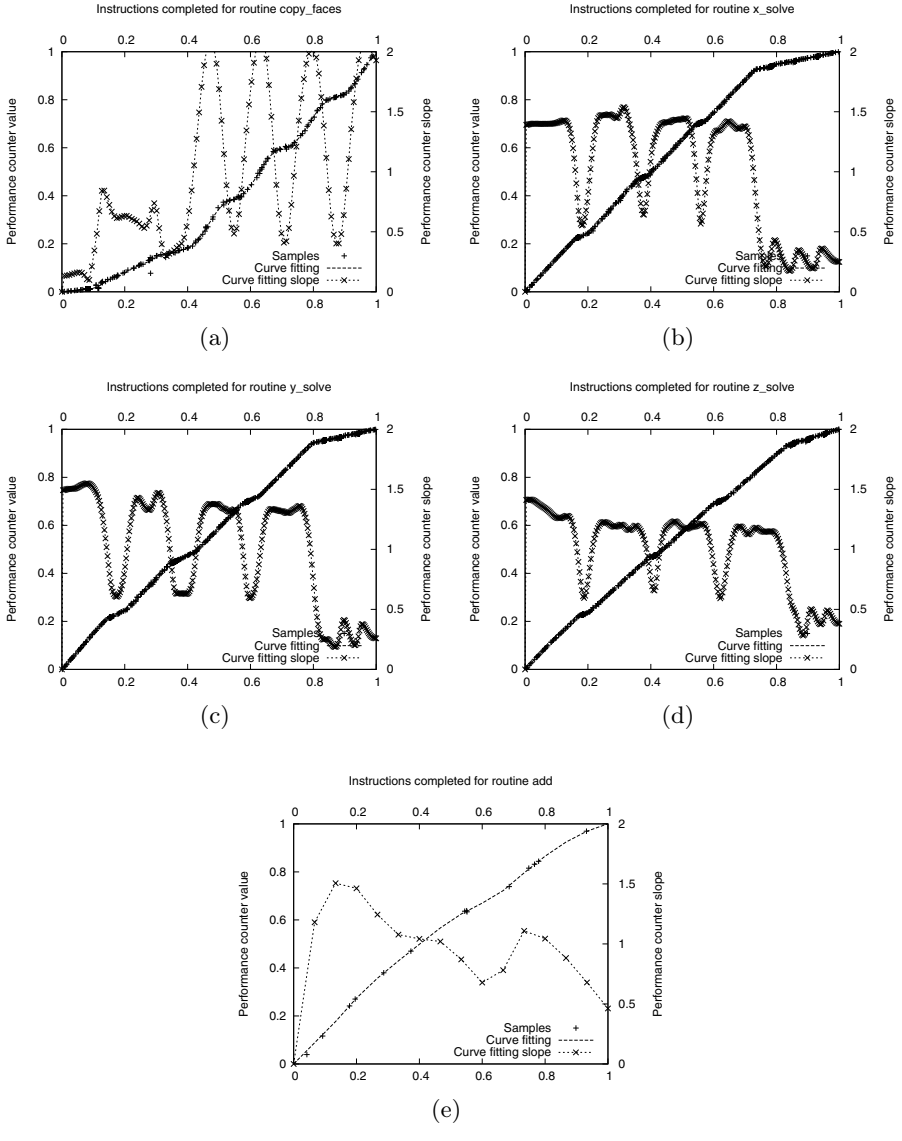


Fig. 4. Evolution of completed instructions for the 1st task in bt.B benchmark in the instrumented routines after applying the folding mechanism

4.1 NAS bt.B

Previous experience on this benchmark has shown that the interesting routines are: `copy_faces`, `x_solve`, `y_solve`, `z_solve` and `add`.

Figures plotted in Figure 4 show the evolution of the completed instruction performance counter among each of the five routines in the first task. The x-axis represent the time (normalized from 0 to 1) in the routine, the left y-axis is used to show the value of the counter (normalized from 0 to 1) within the routine and the right y-axis is used to range the slope of the interpolation. Each plot presents three types of information:

- Samples gathered during instrumentation and folded into the chosen iteration and within the user functions that were taken. Samples are shown by crosses.
- Interpolation of the gathered samples. It is shown by a segmented line.
- Slope of the interpolation of the gathered samples. It is shown by a segmented line with crosses.

The slope of the interpolation facilitates the location of hot-spots (or even cold-spots). For example, in Figure 4(a) three different behaviors exist. First, a small section, which lasts about the first 10% of time, aggregates completed instructions very slowly. Then comes a region with a higher instruction completion rate. And finally, a long region that has four peaks separated by valleys. Looking at the source code of the routine, we find that this behavior corresponds with the execution of a loop in the subroutine `compute_rhs` that is executed four times.

Routines `x_solve`, `y_solve` and `z_solve`, which are shown in Figures 4(b), 4(c) and 4(d) respectively, present a very similar pattern. There is a region that lasts approximately the 80% of time with four peaks that accumulate more than the 90% of the completed instructions. The remaining count of completed instructions come from the remaining 20% of time.

Finally, Figure 4(e) shows the behavior of the `add` routine. It starts by accumulating a high number of completed instructions and then it decreases slowly. At 75% of the routine time it increases again to decrease at the end of the routine.

4.2 Alya

Alya developers helped us locating the most time consuming routines. These routines are: `nsi_elmope`, `gmrpls` and `cgrpls`. We focus on the second task because the parallel computation is done in all tasks except the first, which synchronizes the parallel computation.

Figures plotted in Figure 5 show the evolution of the stalled cycles performance counter among each of the three routines in the second task. The stalled cycles performance counter provides information of the amount of time that the CPU has been waiting for resources to execute an instruction. Such counter is useful to locate code region with bottlenecks that prevents the application going at full speed.

Although there are three instrumented routines, one of them is executed twice in an iteration and thus we present two different plots for it (one for each execution). As seen in the BT example, x-axis represent the time in the routine, the

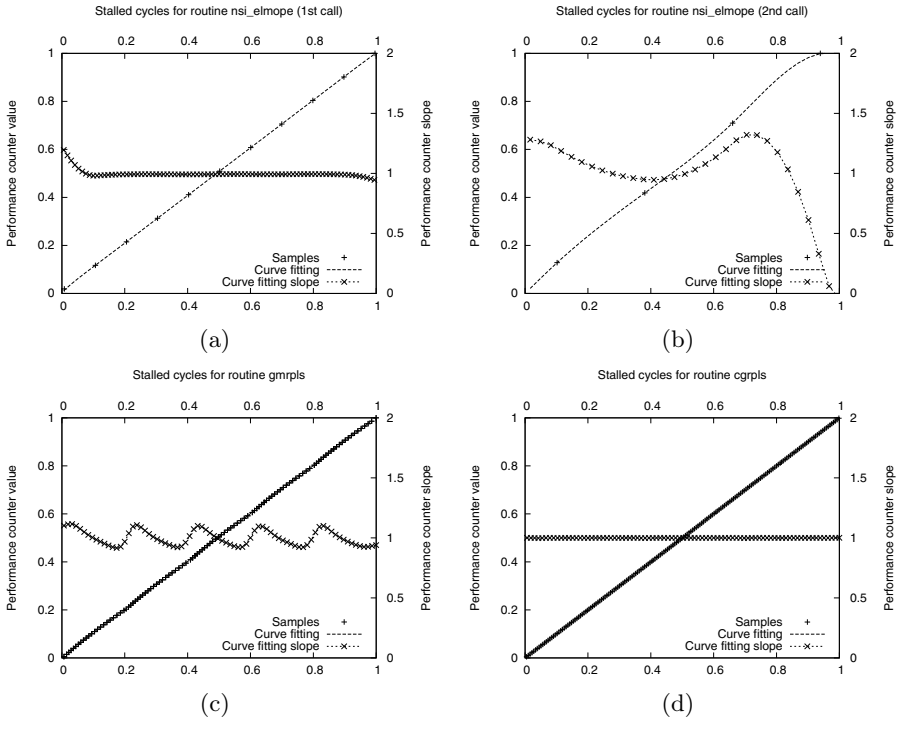


Fig. 5. Evolution of stalled cycles for the 2nd task in the Alya in the selected routines after applying the folding mechanism

left y-axis ranges the value of the counter within the routine and the right y-axis delimits ranges the slope of the interpolation. Each plot presents three types of information: samples gathered, interpolation and the slope of the interpolation.

First of all, it is noticeable that the two calls of `nsi_elmope` behave differently. This routine does the assembly of the continuity and momentum matrices. Regarding its performance, the first call plot is shown in Figure 5(a). The plot presents a steady behavior across the execution except for the margins. However, the second call, which is shown in Figure 5(b), accumulates most of the cycles stalled at the beginning of the run and it suddenly decreases at the 80% of the routine after a peak at that position.

The `gmrpls` routine implements an iterative gmres solver for nonsymmetric matrices. We see in Figure 5(c) that the slope of the evolution of the performance counter behaves as a wave with small amplitude and with a wavelength of 20% of the routine duration.

Finally, the plot in Figure 5(d) shows the performance behavior of the the routine `cgrpls`, which implements a conjugate gradient method. It behaves steadily across the whole execution.

5 Related Work

In this section we summarize the approaches done to combine both instrumentation and sampling mechanisms in the performance analysis area, and how they differ with our work.

The widely known gprof [10] exploits both sampling and instrumentation mechanisms to emit functions call count and estimated spent time. Gprof requires the application to be compiled with a special flag that is responsible for instructing the compiler to add counting monitors in the user routines. Gprof also uses sampling mechanisms to attribute time to the user routines during the execution. The visualization tool echoes to the standard output the fraction of time spent when a routine directly called another one and the number of invocations for each routine. The main difference with our work is that gprof is a performance tool based on profiling whereas our tool is based on tracing. Gprof just provides summaries for simple and concrete function metrics. The solution we propose combines information gathered by tracing and instrumentation to generate a trace with timestamped details that makes the analysis more detailed and precise.

The Sun Studio Performance Analyzer [13] comprises a set of tools for collecting and viewing application performance data using tracing and profiling mechanisms. Its collecting tool is able to trace information relative to synchronization calls, heap allocation and deallocation, OpenMP constructs, MPI routines, data-race and deadlock detection, and counting the number of times each instruction was executed. Furthermore, it uses sampling clock-based or hardware counter overflow mechanisms to profile the target application. Its visualization tool is a multifunctional window that presents the data collected in a wide variety of flavors, including: flat routine profile (similar to *gprof*), calling and called routines, link to source and disassembly, and a timeline window. It exploits the sampling to accumulate the execution time for different routines, and, identify which parts of the user program are responsible for cache or floating point inefficiencies. Although being a powerful set of tools, it needs to collect again the performance data if the results are not detailed enough. This is not an issue for the solution we propose if the application matches a set of requirements described in section 3.1.

Azimi, Stumm and Wisniewski present in [**?**] an online performance analysis tool that gathers counter values periodically or after a designated number of hardware counter occurrences. This tool presents the evolution among the selected counters in a timeline and provides accurate information on which micro-architecture components are stressed using a model called Statistical Stall Breakdown. To provide information for all performance counters it multiplexes counters taking advantage of the underlying functionalities provided by the operating system. Although they offer some instrumentation capabilities, their work is just focused on the sampling mechanism to characterize the whole system and not applications.

SimPoint [20] and SMARTS [23] use sampling in a different context so as to accelerate detailed micro-architecture simulations. Their primary goal is to reduce long-running applications down to tractable simulations. The authors of

both frameworks met this goal by statistically sampling the instruction mix of a running serial application to determine where the application spent time. They combine instrumentation and sampling to collect a sequence of instructions each time the sampling mechanism fires up. The result they obtain is a collection of instruction traces related to each sampling point. These traces will be simulated independently to get detailed performance on the simulated micro-architecture. Our approach, however, combines performance information gathered at different timesteps into a single timestep and the coarse grain sampling produces low overhead penalty in the application run.

A tracing package with some sampling capabilities is presented in [16]. Its functionality is closely related to SimPoint but combining pSiGMA [18] and DynInst [7] instrumentation. Their approach works identifying the timesteps of the application using special instrumentation calls and fully instrumenting the application using the SiGMA toolkit. A DynInst-based tool instruments the special calls added to check whether the executed timestep matches a list of timesteps given by the user to enable or to disable the pSiGMA instrumentation. The fully instrumentation inflicts large overhead penalty when gathering performance data, although it can be greatly reduced by sampling a small set of iterations. Our approach, however, is to take few samples along the application execution and then construct an ideal iteration from the data collected.

6 Conclusions and Future Work

We have explored the possibility of combining both sampling and instrumentation mechanisms to generate more detailed traces.

Our main contribution is the design and implementation of a process called *folding* that provides detailed performance information using coarse grain sampling. It provides detailed performance information at three different levels: iteration, user routines and running bursts. It is suitable for applications based on iterative methods, which is a representative part of the applications found in HPC environments. For the rest of applications, a finer sampling resolution should be chosen to obtain a trace with more details.

To demonstrate its utility we have extended the MPItrace instrumentation package with a sampling mechanism using hardware counters to record performance information across the whole execution. The combination of instrumentation and sampling produces traces containing performance information from both instrumented and uninstrumented regions of code.

The additional performance information provided by the sampling, and the ability to project it into representative areas, brings a new methodology into play that has proven useful for the detailed analysis of real applications. In particular, we have shown detailed performance analysis for the representative routines of the NAS BT benchmark and the Alya application.

Finally, we believe that some of issues in the methodology are still open.

First, provide a way to automatically determine which are the interesting and representative user routines to be analyzed without using a profiler. Samples

could also store the stack trace, in addition to the performance counter values and the address of the instruction that triggered the sample, in order to provide information of the executed routines.

Second, automatically identify the application iterations structure using periodicity or application structure detectors like [8] and [9]. These detectors work directly with Paraver traces and the reported information could be used as input of the folding mechanism to delimit the regions to be folded instead of adding manually events into the application source code.

Finally, we would like to study the impact of the sampling rate and the number of sampled timesteps on the folding results.

Acknowledgements

We would like to acknowledge José María Cela and Pierre Lafortune for their insightful comments on contouring algorithms and the implementation of the Kriging algorithm. This work is granted by the IBM/BSC MareIncognito project and by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) (contract no. TIN2007-60625).

References

1. Intel trace collector and analyzer, `http://www.intel.com/cd/software/products/asmo-na/eng/306321.htm`
2. MPItrace instrumentation package, `http://www.bsc.es/plantillaA.php?cat_id=492`
3. NAS parallel benchmark suite, `http://www.nas.nasa.gov/Resources/Software/npb.html`
4. Azimi, R., Stumm, M., Wisniewski, R.W.: Online performance analysis by statistical sampling of microprocessor performance counters. In: ICS '05: Proceedings of the 19th annual international conference on Supercomputing, pp. 101–110. ACM, New York (2005)
5. Bézier, P.: Numerical Control. Mathematics and Applications. John Wiley and Sons, London (1972) Translated by: Forrest, A.R., Pakhurst, A.F.
6. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. 14(3), 189–204 (2000), `http://icl.cs.utk.edu/papi`
7. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. Int. J. High Perform. Comput. Appl. 14(4), 317–329 (2000), `http://www.dyninst.org`
8. Casas, M., Badia, R.M., Labarta, J.: Automatic Structure Extraction from MPI Applications Tracefiles. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 3–12. Springer, Heidelberg (2007)
9. González, J., Giménez, J., Labarta, J.: Automatic Detection of Parallel Applications Computation Phases. In: IPDPS'09: 23rd IEEE International Parallel and Distributed Processing Symposium (2009)
10. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. In: SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, pp. 120–126. ACM, New York (1982)

11. Hempel, R.: The MPI standard for message passing. In: HPCN Europe 1994: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, London, UK, vol. II, pp. 247–252. Springer, Heidelberg (1994)
12. Houzeaux, G., Vázquez, M., Grima, R., Calmet, H., Cela, J.M.: Experience in parallel computational mechanics on marenostrum (2007)
13. Itzkowitz, M.: Sun studio performance analyzer
14. Moore, S.V.: A comparison of counting and sampling modes of using performance monitoring hardware. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J., Hoekstra, A.G. (eds.) ICCS-ComputSci 2002. LNCS, vol. 2330, pp. 904–912. Springer, Heidelberg (2002)
15. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12(1), 69–80 (1996)
16. Odom, J., Hollingsworth, J.K., DeRose, L., Ekanadham, K., Sbaraglia, S.: Using dynamic tracing sampling to measure long running programs. In: SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, p. 59. IEEE Computer Society, Los Alamitos (2005)
17. Pillet, V., Labarta, J., Cortés, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. Transputer and occam Developments, 17–32 (April 1995), `http://www.bsc.es/plantillaA.php?cat_id=485`
18. Sbaraglia, S., Ekanadham, K., Crea, S., Seelam, S.: pSIGMA: An infrastructure for parallel application performance analysis using symbolic specifications. In: Proceedings of EWOMP'04 (October 2004)
19. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)
20. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. SIGOPS Oper. Syst. Rev. 36(5), 45–57 (2002)
21. Trochu, F.: A contouring program based on dual Kriging interpolation. Engineering with Computers 9(3), 160–177 (1993)
22. Wolf, F., Mohr, B.: Kojak - a tool set for automatic performance analysis of parallel applications. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 1301–1304. Springer, Heidelberg (2003); Demonstrations of Parallel and Distributed Computing
23. Wunderlich, R.E., Wenisch, T.F., Falsafi, B., Hoe, J.C.: Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In: ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture, pp. 84–97. ACM, New York (2003)