# Chapter 9
# Folding: Detailed Analysis with Coarse Sampling

**Harald Servat, Germán Llort, Judit Giménez, Kevin Huck, and Jesús Labarta**

**Abstract** Performance analysis tools help the application users to find bottlenecks that prevent the application to run at full speed in current supercomputers. The level of detail and the accuracy of the performance tools are crucial to completely depict the nature of the bottlenecks. The details exposed do not only depend on the nature of the tools (profile-based or trace-based) but also on the mechanism on which they rely (instrumentation or sampling) to gather information.

In this paper we present a mechanism called folding that combines both instrumentation and sampling for trace-based performance analysis tools. The folding mechanism takes advantage of long execution runs and low frequency sampling to finely detail the evolution of the user code with minimal overhead on the application. The reports provided by the folding mechanism are extremely useful to understand the behavior of a region of code at a very low level. We also present a practical study we have done in a in-production scenario with the folding mechanism and show that the results of the folding resembles to high frequency sampling.

## 9.1 Introduction

Application users are typically delighted when they are granted access to a new supercomputer because they expect their applications to run at a faster pace than before. Although that each new supercomputer reports faster results than their predecessors, it is unquestionable that user applications only reach a portion of the peak performance of the supercomputer.

H. Servat (✉) · G. Llort · J. Giménez · K. Huck · J. Labarta
Barcelona Supercomputing Center, Universitat Politècnica de Catalunya, c/Jordi Girona, 31, 08034 Barcelona, Catalunya, Spain
e-mail: harald.servat@bsc.es; german.llort@bsc.es; judit.gimenez@bsc.es; kevin.huck@bsc.es; jesus.labarta@bsc.es

Performance analysis tools aim to explain to the user the reasons why his or her application cannot reach the supercomputer's peak performance to eventually optimize the application and increase its performance. Nowadays we can classify performance tools by the amount of data gathered at runtime: profile-based and trace-based. While profile-based tools keep timeless performance data of the execution run by aggregating the collected data, trace-based tools generate a sequence of timestamped events that report the progression of the application. In addition, performance tools can use two different methods to gather performance metrics sampling and instrumentation. One the one hand, sampling relies on periodic signaling to gather performance metrics, on the other hand, instrumentation injects code to specific locations of an application to emit the metrics. To increase the details of such methods the user may decrease the period of the sampling or instrument additional code locations, but with the extra overhead drawback. An alternative is to combine both instrumentation and sampling as in Extrae [4], TAU [15] or Scalasca [19].

In this paper we describe folding [13, 14], a mechanism for trace-based performance tools, that provides high level of detail using Paraver [12] traces. The folding mechanism combines sampled and instrumented information gathered from the sample application in order to produce more detailed results than its original form in the Paraver tracefile. Moreover, folding benefits from long running applications because it can be combined with high periodicity sampling to produce detailed results with low overhead cost. We have combined the folding mechanism with a density-based clustering tool [5, 6] to develop a framework that automatically provides characteristics of the most time-consuming computation regions in an application. This allows the analyst to focus on potential performance issues on specific code locations that represent the relevant part of the application.

The rest of this paper is structured as: Sect. 9.2 reviews both sampling and folding concepts and describes how the performance counters and the callstack information is folded. Section 9.3 shows the folding results for a set of analyzed applications. We compare the quality of the folding results with fine grained sampling results in Sect. 9.4. Finally, we discuss related and future work in Sects. 9.5 and 9.6, respectively.

## 9.2 Folding: Instrumentation and Sampling

We extended the Extrae [4] instrumentation package with sampling capabilities. The sampling mechanism can either rely on the PAPI middleware (`PAPI_overflow`) or the regular operating system alarms. The sampling handler is responsible for gathering hardware performance counters and a segment of the whole callstack at the sample point. We depict how the overhead increases the execution time of the application when using smaller sampling periods in Fig. 9.1. Naturally, the more samples the library gathers the more the application becomes perturbed due to the intrusion of the library. For example, when the sampling period is $50 * 10^3$

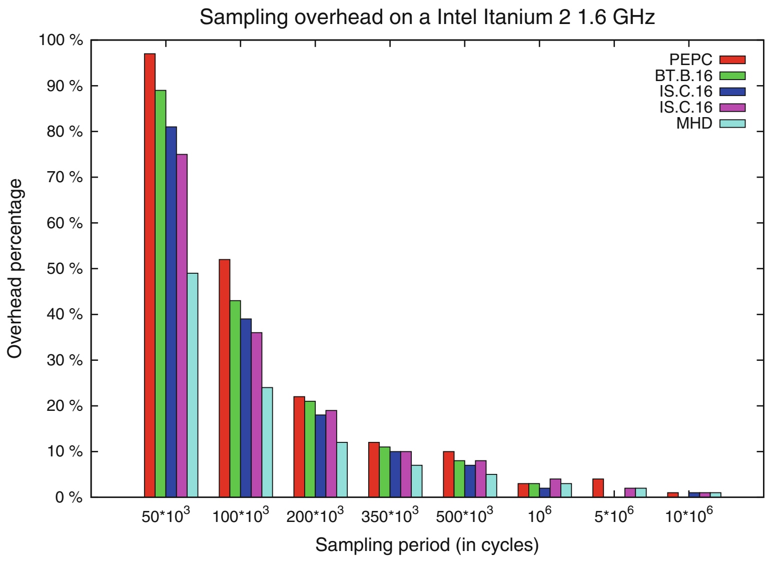


**Fig. 9.1** Overhead of the sampling mechanism at different sampling periods when using different MPI applications on a Intel Itanium2 supercomputer

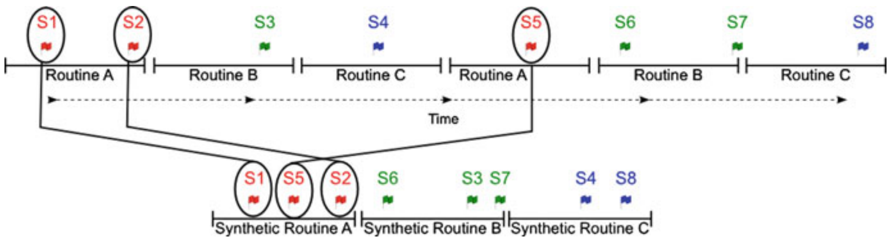


**Fig. 9.2** Example of the folding mechanism. The *top* timeline shows the samples belonging to the different routines. In the *bottom* figure, we plot how the folding maps the samples on the different synthetic regions. We have outlined the mapping for the *Routine A* which involves the samples labeled as *S1, S2* and *S5*

cycles the application can take up to 100 % more time to run. This large overhead makes the fine grained sampling not desirable because of the large impact it has on the application.

The folding mechanism takes benefit of long running applications (which is very common in the scientific computing) to combine sampled information from different regions into a single synthetic region. This mechanism combines instrumented and sampled information to increase the details of the instrumented regions. Within the folding process, each instrumented and sampled information plays a role. While instrumented information is used to delimit regions, sampled information determines how the performance behavior evolves within the region it belongs. Consider the time-line shown in Fig. 9.2 where an iterative application executes three instrumented routines named A, B, C. Each single flag represents a taken sample. The folding mechanism creates three synthetic regions (one per routine) and then maps every sample into those regions according to the routine that was being executed at the sample point. The result is shown below in the same Figure, where the reader can observe that the mapped samples within the synthetic region preserve their position in respect of their original region instead of their temporal ordering.

### 9.2.1 The Hardware Counters

The folding results, regarding the hardware counter information, are plotted in Fig. 9.3. This plot shows the evolution of the completed instructions counter using the folding samples in a computation region of the NAS BT benchmark [11]. The folded samples belonging to the selected region are normalized and shown as red crosses. A red cross in the point (X,Y) means that a sample was taken at the X % from the beginning of the region and counted the Y % of the total metric in the region. In both figures, we can infer temporal phases with different performance characteristics also noting their sequence and duration by solely using the folded samples.

Once we have the cloud of samples, we perform a polynomial adjustment. The adjustment serves three purposes: (a) as an analytical model we can compute its derivative and, from the derivative, we can compute the instantaneous rates, (b) we sample the result to reintroduce the folded metrics as synthetic events into the tracefile at a user requested rate, and (c) it serves also as a noise reduction mechanism.

We explored several approaches to compute the polynomial adjustment: polynomial fitting, Bézier curves [2] and Kriging [18] interpolation. Polynomial fitting requires choosing the grade of a polynomial which 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 also fitted well on our tests except on the stationary points. Finally, the Kriging interpolation, which is a general version of the Bézier curve, 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.

In Fig. 9.3, the contouring results and the instantaneous rate are shown in the plot as a green and blue lines, respectively. Now comparing the information derived from the adjustment and its derivative, we can easily identify four different phases by their MIPS rate. Each phase involves a period in which the routine runs really fast (at more than 6,000 MIPS) and follows a period where the routine runs slower (at 2,000 MIPS, approximately).

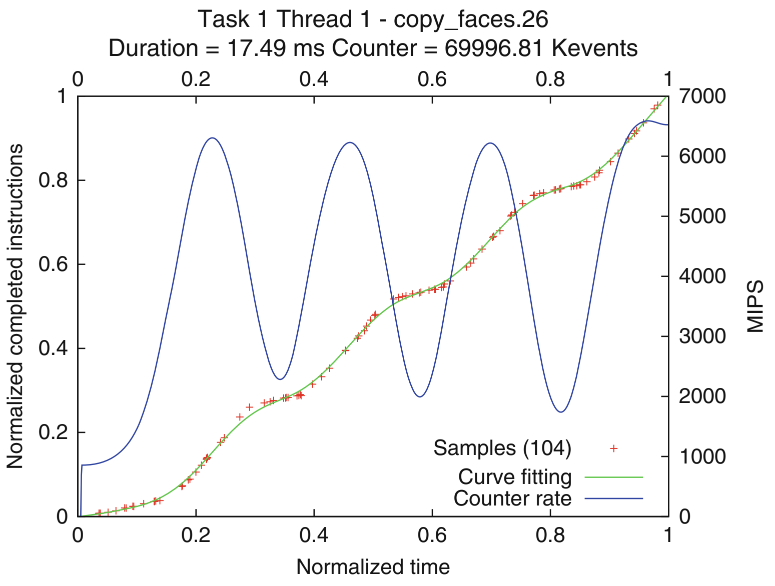


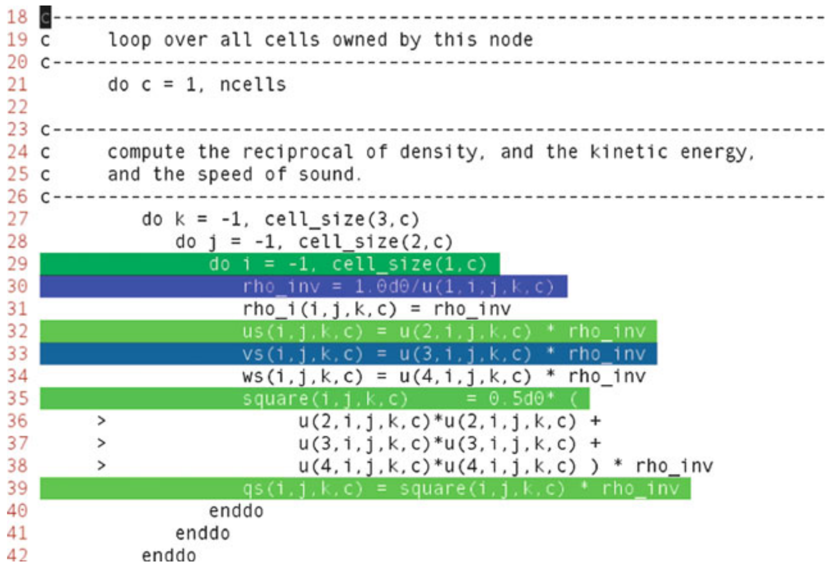
**Fig. 9.3** Folding results of the instructions counter for a computation region in the NAS BT benchmark when executed in a SGI Altix 4700 machine

### *9.2.2 The Callstack*

The folding mechanism is also applied to the sampled callstack information if the application contained debugging information during the execution. In contrast to the folded hardware counters, folding the callstack information does not benefit from an analytical model like Kriging. The folded callstacks are emitted to the tracefile and can be used to display which code region was being executed at a certain time in the synthetic regions. Although this method can be sensitive to noise, it provides an approximate location where to start looking in the application code. With such information, a time-based analysis tool like Paraver can create correlations between the folded callstack information and any metric. Such correlation can subsequently stored and then display those metrics in GVIM (a *vim*[1] editor using a *GNOME*[2] front-end) using an add-on we built using the GVIM scripting mechanism. We show a screenshot of this add-on on Fig. 9.4. In this Figure, the lines that achieved the lowest and the highest MIPS are annotated in colors, ranging from a minimum in green to a maximum in blue.

[1]http://www.vim.org

[2]http://www.gnome.org



```
c-----------------------------------------------------------------------
c     loop over all cells owned by this node
c-----------------------------------------------------------------------
      do c = 1, ncells

c-----------------------------------------------------------------------
c     compute the reciprocal of density, and the kinetic energy,
c     and the speed of sound.
c-----------------------------------------------------------------------
         do k = -1, cell_size(3,c)
            do j = -1, cell_size(2,c)
               do i = -1, cell_size(1,c)
                  rho_inv = 1.0d0/u(1,i,j,k,c)
                  rho_i(i,j,k,c) = rho_inv
                  us(i,j,k,c) = u(2,i,j,k,c) * rho_inv
                  vs(i,j,k,c) = u(3,i,j,k,c) * rho_inv
                  ws(i,j,k,c) = u(4,i,j,k,c) * rho_inv
                  square(i,j,k,c)     = 0.5d0* (
     >                 u(2,i,j,k,c)*u(2,i,j,k,c) +
     >                 u(3,i,j,k,c)*u(3,i,j,k,c) +
     >                 u(4,i,j,k,c)*u(4,i,j,k,c) ) * rho_inv
                  qs(i,j,k,c) = square(i,j,k,c) * rho_inv
               enddo
            enddo
         enddo
```

**Fig. 9.4** GVIM capture showing the MIPS within the editor according to the stats captured at runtime and the results of the folding mechanism

## 9.3 Example of Usage

To exemplify the usage, we will show the study we did of the Code_Saturne [3] application, which also belongs to the PRACE Benchmark Suite [16]. The application was executed on MareNostrum, a 10,240 core supercomputer based on two dual-core PowerPC 2.3 GHz processors per blade interconnected with a Myrinet network. The application was compiled with the IBM XL Fortran compiler version 12.1 using `-O3 -qstrict` and the experiment ran 200 time-steps using 32 cores with a sampling period of 100 ms and MPI instrumentation.

We used a clustering tool [5, 6] that characterizes computation regions according to their metrics characteristics to find the application structure. The results for the characterization are shown in Fig. 9.5. Each dot within the scatterplot represents a computation region (i.e., a region within an MPI exit point and the consecutive MPI entry point). We classified every computation region by two performance metrics of the region: its executed instructions and its IPC (instructions per cycle). This classification shows on one hand the total amount of work done by a portion of code and the speed achieved to execute that amount of work. After the classification, the computation regions are grouped (colored in the plot) together by their distance to each other using a density-based algorithm and the grouping information is sent back to the tracefile for further analyses. From all the resulting groups we focused on the more relevant ones. Concretely, we will focus on those computation regions that last more than 50 ms, and from them we will focus on Cluster 1 because

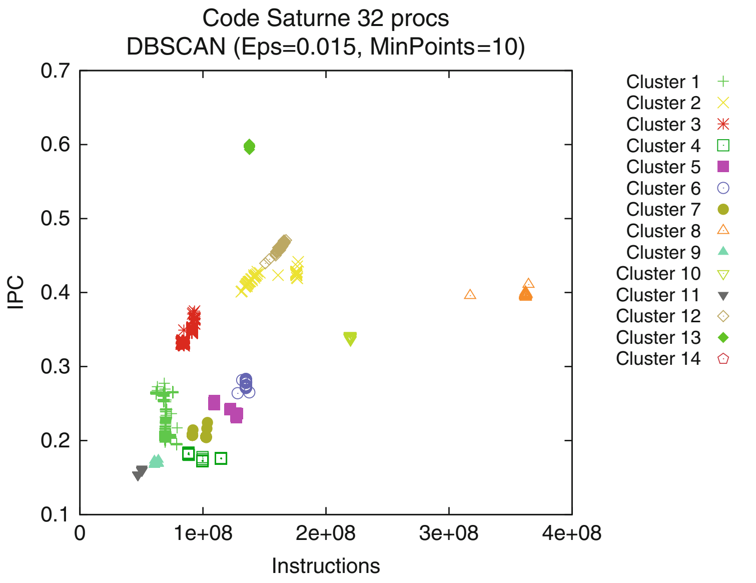


**Fig. 9.5** Scatterplot shows the grouping of the different computation regions of the Code_Saturne application. The Y-axis shows the measured IPC whereas the X-axis the committed instructions

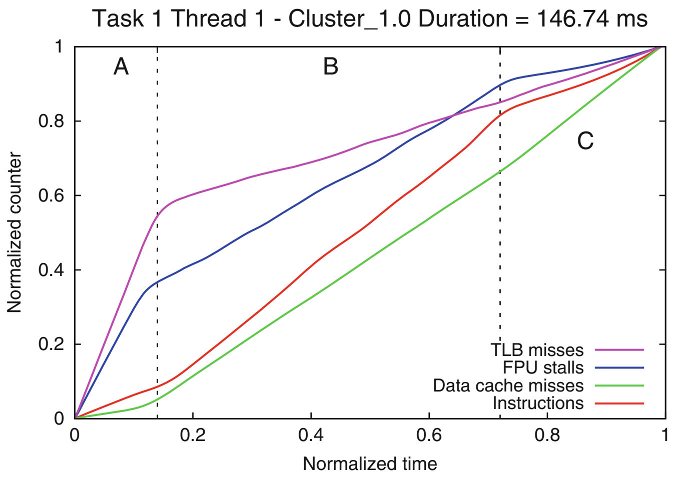


**Fig. 9.6** Plot depicting the evolution of several performance counters (namely: instructions, data cache misses, FPU stalls and TLB misses) within Cluster 1 in the Code-Saturne application

it has more than 20K occurrences and accounts more than the 35 % of the total computation time.

To study the internal behavior of Cluster 1, we applied the folding mechanism on these regions. Due to space restrictions, we have summarized the results we obtained in Fig. 9.6. In this figure we show the evolution of four different performance

**Table 9.1** Average values for different metrics in Code_Saturne code divided by phases using the slope of the instructions slope

| Computation region | Cluster 1 | | |
|---|---|---|---|
| Phase name | A | B | C |
| Location in `gradrc.f90` | 751–753 | 786–802 | 972–985 |
| Duration[a] | 20.54 | 84.82 | 41.08 |
| MIPS | 300 | 600 | 300–450 |
| Data cache stall cycles[b] | 400 | 1,700 | 2,000 |
| TLB miss stall cycles[b] | 160 | 20 | 20 |
| FPU stall cycles[b] | 550 | 160 | 75 |

[a]In milliseconds
[b]In millions per second

counters (instructions, data cache misses, FPU stalls and TLB misses) within Cluster 1 using the interpolation results of the folded samples. We have manually divided the plot by the slope of the instructions counter rate to divide different computation phases (labeled as A, B and C in this Figure) within the same computation region. In Table 9.1 we show different metrics and the code location for each phase.

According to the metrics we extracted from the Paraver tracefile we obtained that `Cluster 1` ran at approximately 472 MIPS which is very bad for the MareNostrum processor because it represents that it achieved a CPI (cycles per instruction) of 4.87. Inside the cluster we can differentiate three phases considering the instructions slope. First phase goes from the beginning to 0.15, the second goes from 0.15 to 0.75 and the final goes from 0.75 to the end. According to the information we have obtained during the execution this cluster is mainly executing the routine contained in `gradrc.f90`. After analyzing the folded callstack, we can relate them to three different loops at the following lines at lines 751–753, 786–802 and 972–985, respectively.

The loop in lines 751–753 shows a CPI between 7 and 9. This loop accesses seven different vectors and performs a large number of floating point computations. We observed that this phase ran at 300 MIPS and that the processor was stalled by TLB misses and by lack of FPU resources. The second loop contains lots of pointer indirections and some floating point instructions. We observe that 1,700 Mcycles out of 2,300 Mcycles (i.e., the processor frequency) are stalled due to data cache misses, however the processor is capable of executing instructions at 600 MIPS. Last but not least, we see that the last pointed loop has a worse performance behavior in terms of stalled cycles due to cache misses resulting in a worse MIPS rate (ranging from 300 to 450).

We unsuccessfully tried to improve the performance of the application. We splitted the first loop into three loops to prevent the cache to hold all the vectors referenced, however after applying this modification the average duration of Cluster 1 increased by 5 ms. Changing the second loop would need changing the face numbering algorithm to increase the locality of the accessed data and/or rethinking this part of the code to reduce the number of indirections. Also, it would

**Table 9.2** Experiment setup for the quality study

| | Application execution setup | |
|---|---|---|
| Application name | BT.B | |
| Sampling mode | Detailed | Coarse |
| Sampling period | 50 Kcycles | 10 Mcycles |
| Samples per second | 32,000 | 160 |
| Sampling overhead | 89 % | 2 % |
| Number of iterations | 1 | 200 |
| Total number of samples per task | 5,661 | 5,445 |

be interesting to reduce the number of accesses within the loop because there are currently 11 arrays accessed. Finally, we applied blocking and splitting techniques to the third loop to increase the locality of the data accesses. However, when applying blocking with a block size of 128 elements, it made Cluster 1 3 % slower whereas splitting the loop made Cluster 1 a 20 % slower, mainly due to the increase of the TLB misses.

## 9.4 Validation of the Results

We validated and studied the quality of the folding mechanism by using the BT.B benchmark from the NAS MPI Parallel Benchmark Suite 3.2 on a SGI Altix machine with Intel Itanium 2 processors running at 1.6 GHz. Although this benchmark is heavily optimized, some computing regions of BT.B show a non-uniform behavior (as seen in Fig. 9.3) making it a nice candidate to study. The Kriging contouring algorithm used in the folding mechanism behaves as a low-pass filter, so with the higher number of samples being interpolated the interpolation results are more detailed. Because of this, the experiments are focused on validating how the folded and interpolated results resemble finer sampled metrics. Details regarding the experimentation are shown in Table 9.2.

The first experiment is intended to compare the shape of the hardware counter metrics in high frequency sampled traces and low frequency folded sampled traces. We did the comparison with two of the most time consuming computation regions from the benchmark. To facilitate the reading, each computation region is named according to the routine to which it belongs. We compare one time-step of the high frequency trace to the folded results of all iterations. The comparison is done using different counters available on the chosen processor, namely committed instructions, executed floating point operations and branches, and L2 cache accesses, hits, and misses.

The Kriging implementation is limited to return a vector of equidistant points which prevents us to get the value of the interpolation at any arbitrary point. To perform the study, we compared the results between the samples of the reference

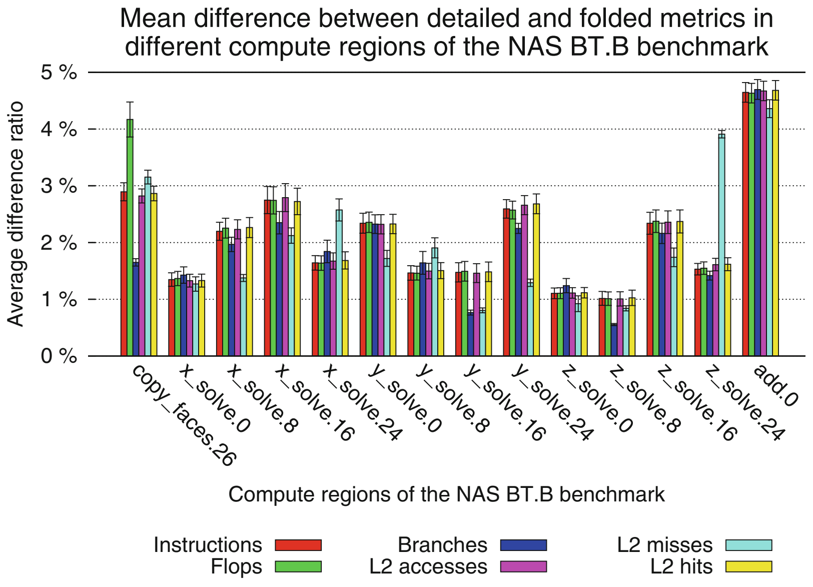


**Fig. 9.7** Study of absolute mean difference ratios and square mean errors on different applications comparing high detail sampling and folded results on the NAS BT.B benchmark

time-step and the folded samples by interpolating both of them in the same number of points and then we compute the absolute mean difference and the square mean error by comparing every resulting slice.

Results of the comparison are shown in Fig. 9.7. The X-axis on the plot depicts the representative computation regions in the NAS BT benchmark, whereas the Y-axis represents the percentage of variation between the detailed sampling and the folded results using coarse sampling and the square mean error by the bars and the whiskers respectively. In the plot we observe mean differences up to 5 % for every pair performance counter and computation region. The coarse-grained sampling generates approximately the same number of samples for the whole run as the reference iteration because of the number of the time-steps iterated on the coarse-grain execution is the ratio between the two sampling frequencies. So we conclude that using scattered samples on long execution runs we can obtain a good approximation of the performance metrics without harming the gathered performance metrics due to the sampling overhead.

In the second experiment we ran measured how the number of folded samples influences the absolute mean difference shown in the previous experiment. To carry out the comparison we took as a reference the highly detailed sampled information of the longest computation regions of the BT.B benchmark from the previous experiment, and then we computed the mean difference varying the number of samples folded on the coarse-grained execution.

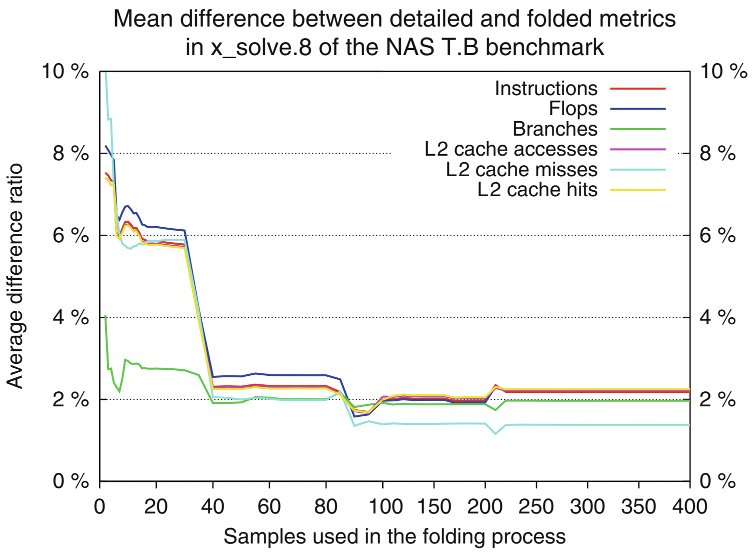


**Fig. 9.8** Evolution of the mean difference between highly detailed sampling and folded results in a BT.B compute region of the x_solve routine

The plot in Fig. 9.8 shows that the absolute mean difference varies depending on the number of samples folded in the region. This plot shows that the mean absolute difference decreases as the number of samples increase. Precisely, the number of samples on which the difference is below 5 % is about 40 for the computation region we present and using more samples just leads to marginal differences. So we can conclude that there is no need for extremely long executions to get similar results between folding coarse grain samples and using detailed sampling, although this may vary on the application.

## 9.5 Related Work

The gprof [7] profiler uses sampling and instrumentation to gather function call counts and estimate the time spent per function. This profiler requires the application to be compiled and linked with a special flag that instructs the compiler adding monitoring to count the number of calls to user routines, while it uses sampling to accumulate time to the user routines. This tool provides summaries for simple metrics, whereas the folding mechanism combines both sampling and instrumentation providing much more details resulting in more helpful analysis.

HPCToolkit [17] uses sampling with trampoline optimizations to provide callstack details. HPCToolkit can also show sampled callstack data in a time-line using a recently added *hpctraceview*. Our mechanism also provides instrumentation and sampling data in a Paraver time-line but is also capable of combining the performance data within the tracefile to provide highly detailed synthetic regions. The synthetic information is reintroduced into the Paraver tracefile so all the metrics can be used together to correlate them easily.

The Sun Studio Performance Analyzer [8] is a set of tools for collecting and viewing application performance data using tracing and profiling mechanisms. These tools are able to instrument synchronization calls, heap allocation and deallocation routines, parallel regions and constructs, and can also sample the application to profile it. Although being a powerful set of tools, the experiments needed to be repeated if the results were not detailed enough. The work we propose is designed to produce highly detailed performance metrics avoiding repetitive data collection.

Azimi, Stumm and Wisniewski are the authors of [1] where they present an on-line performance analysis tool that gathers counter values periodically. This tool displays the evolution among the selected counters in a time-line breakdowns the cycles-per-instruction using a model called Statistical Stall Breakdown. Although they offer some instrumentation capabilities, their work is just focused on the sampling mechanism to characterize the whole system instead of applications.

The TAU framework has recently added sampling capabilities in its measurement system [10]. And, although they gather performance counter information, their work is mainly focused on instrumentation cooperating with the sampling to augment the TAU profiles with PC callstack information instead of increasing the details of the performance counters. The cooperation to increase their profiles is based on creating keys to the application callstack and to the TAU event stack during the application execution to reduce the information presented to the user in order to produce a summarized view.

Finally, the Scalasca tool has also been extended using sampling. As in TAU, their developers put the major effort to provide information about application routines instead of providing performance counters. Scalasca provides accurate time metrics by subtracting the time in MPI calls of an user routine for a sampling period when PMPI instrumentation is turned on. Also in Scalasca, as in HPCToolkit, trampolines are used to replace the returning addresses in the callstack to reduce the overhead in the unwinding process.

## 9.6 Conclusions and Future Directions

We have evaluated the quality of the results of using low frequency sampling and applying the folding mechanism compared with high frequency sampled data. The results show good resemblance (less than 5 % of difference) between them, even using only a fraction of the samples of the whole run. It makes low frequency sampling and folding a good alternative to high frequency sampling without penalizing the application. We have shown an example of utilization of the folding when combined with the clustering. Yet the example we provided was not improved, we have shown successful studies and optimizations in [14].

Future work could include working on a better correlation between performance counters at this finer detail to thoroughly describe the performance behavior and also to locate hot and cold spots within the code. Another interesting future topic would be to perform an on-line study of the application using the work described in [9]. The combination of these two mechanism would provide detailed performance information as the application executes, showing the performance at different application stages and stop when enough samples have been gathered and study whether the sampling periodicity matches any period of the computation bursts.

**Acknowledgements** This work is granted by the IBM/BSC MareIncognito project and by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) under Contract No. TIN2007-60625.

## References

1. Azimi, R., et al.: 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). doi:http://doi.acm.org/10.1145/1088149.1088163
2. Bézier, P.: Numerical Control. Mathematics and Applications. Wiley, London (1972). Translated by: A.R. Forrest and Anne F. Pakhurst
3. Code Saturne. http://research.edf.com/research-and-the-scientific-community/softwares/code-saturne/introduction-code-saturne-80058.html. Accessed July 2011
4. Extrae Instrumentation Package. http://www.bsc.es/paraver. Accessed August 2012
5. González, J., et al.: Automatic detection of parallel applications computation phases. In: IPDPS'09: 23rd IEEE International Parallel and Distributed Processing Symposium, Rome, Italy. IEEE Computer Society, Piscataway (2009)
6. González, J., et al.: Automatic evaluation of the computation structure of parallel applications. In: PDCAT '09: Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies, Hiroshima, Japan. IEEE Computer Society, Hiroshima (2009)
7. Graham, S.L., et al.: 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). doi:http://doi.acm.org/10.1145/800230.806987
8. Itzkowitz, M.: Sun studio performance analyzer. http://developers.sun.com/sunstudio/overview/topics/analyzer_index.html. Accessed August 2012
9. Llort, G., et al.: On-line detection of large-scale parallel application's structure. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 19–23 April 2010, pp. 1–10. doi: 10.1109/IPDPS.2010.5470350. URL:http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5470350&isnumber=5470342 (2010)
10. Morris, A., et al.: Design and implementation of a hybrid performance measurement and sampling system. In: ICPP 2010: Proceedings of the 2010 International Conference on Parallel Processing, San Diego, California (2010)
11. NAS Parallel Benchmark Suite. http://www.nas.nasa.gov/Resources/Software/npb.html. Accessed August 2012
12. Pillet, V., et al.: Paraver: a tool to visualize and analyze parallel code. In: Nixon, P. (ed.) Transputer and occam Developments, pp. 17–32. IOS Press, Amsterdam (1995). http://www.bsc.es/paraver. Accessed July 2011

13. Servat, H., et al.: Detailed performance analysis using coarse grain sampling. In: Euro-Par Workshops (Workshop on Productivity and Performance, PROPER), Delft, The Netherlands pp. 185–198. Springer Berlin, Heidelberg (2009)
14. Servat, H., et al.: Unveiling internal evolution of parallel application computation phases. In: ICPP'11: International Conference on Parallel Processing, Taipei, Taiwan (2011)
15. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). doi:http://dx.doi.org/10.1177/1094342006064482
16. Simpson, A.D., Bull, M., Hill, J.: Identification and categorisation of applications and initial benchmarks suite (2008). http://www.prace-project.eu/documents/Identification_and_Categorisatio_of_Applications_and_Initial_Benchmark_Suite_final.pdf. Accessed July 2011
17. Tallent, N., et al.: Hpctoolkit: performance tools for scientific computing. J. Phys. Conf. Ser. **125**(1), 012088 (2008)
18. Trochu, F.: A contouring program based on dual Kriging interpolation. Eng. Comput. **9**(3), 160–177 (1993)
19. Wolf, F., et al.: Usage of the SCALASCA for scalable performance analysis of large-scale parallel applications. In: Tools for High Performance Computing, pp. 157–167. Springer, Berlin/Heidelberg (2008)