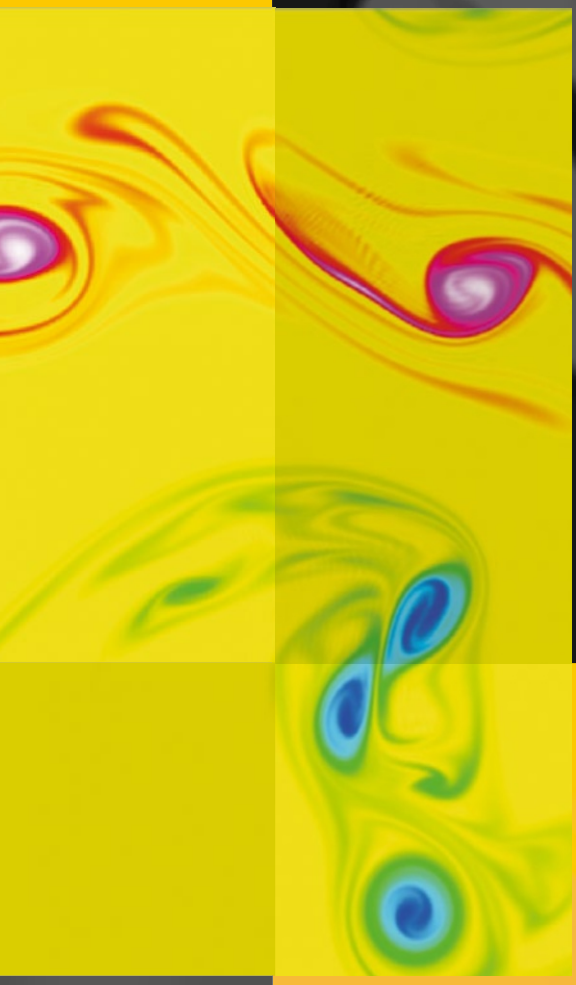
Andreas Knüpfer · José Gracia
Wolfgang E. Nagel · Michael M. Resch
Editors

Tools for High Performance Computing 2013



HLRS

Springer

Tools for High Performance Computing 2013

Andreas Knüpfer • José Gracia •
Wolfgang E. Nagel • Michael M. Resch
Editors

Tools for High Performance Computing 2013

Proceedings of the 7th International Workshop on Parallel Tools for High Performance Computing, September 2013, ZIH, Dresden, Germany

Springer

Editors
Andreas Knüpfer
Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH)
Technische Universität Dresden
Dresden
Germany

José Gracia
Höchstleistungsrechenzentrum Stuttgart (HLRS)
Universität Stuttgart
Stuttgart
Germany

Wolfgang E. Nagel
Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH)
Technische Universität Dresden
Dresden
Germany

Michael M. Resch
Höchstleistungsrechenzentrum Stuttgart (HLRS)
Universität Stuttgart
Stuttgart
Germany

Front cover figure: Numerical simulation of a Kelvin–Helmholtz instability. Illustration by Lars Haupt, Center for Information Services and High Performance Computing (ZIH), 01062 Dresden, Germany

ISBN 978-3-319-08143-4 ISBN 978-3-319-08144-1 (eBook)
DOI 10.1007/978-3-319-08144-1
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014950999

Mathematics Subject Classification (2010): 68M20, 65Y20, 65Y05

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Ongoing research towards specifications and designs for prospective Exascale systems highlights daunting challenges: billion-way parallelism, more levels of parallelism, a very low memory per core ratio, potential for mean-time-between-failures of less than a day, high potential for silent bit errors even in the presence of error-correcting code memories, reduced I/O bandwidths per core, and energy considerations that could require application awareness. Without upcoming disruptive computer hardware, the operation and usage of such systems will require extreme diligence. Research in areas such as parallel programming paradigms, runtime systems, compilers, operating systems, system libraries, and fault-tolerant algorithms can provide answers to some of these challenges, but will not be able to hide all of these effects from application developers.

Such a situation motivates the use of specialized tools for high performance computing (HPC) more than ever. Performance optimizers can aid application programmers in the identification of additional potential for parallelism, identify inefficient use of specific levels of parallelism, and incorporate feedback from energy monitors to identify wasteful resource usage. Debugging and correctness tools could integrate feedback from the operating system and fault tolerant system libraries to highlight the presence and impact of system faults. Simulators and automatic optimizers could identify inefficient data movement patterns, optimize process/thread mappings, or even highlight potential errors from silent memory faults.

This year's International Parallel Tools Workshop in Dresden, the seventh in a series of workshops that started in 2007, highlights how tools must carefully consider these challenges. Presentations on the state of the art in HPC tools both include activities to incorporate advances in programming paradigms, system libraries, and novel information sources, as well as new concepts for data analysis, presentation, correlation, prediction, simulation, and visualization in order to ensure that application developers will not be overwhelmed by the massive amounts of data

that tools can generate even for the Petascale era. The contributions of this year's workshop include the following topics:

- Performance analytics that maximize insight of performance data
- A workflow that captures the essential behavior of existing parallel applications and automatically generates benchmark codes with the ability to extrapolate said behavior at Exascale levels
- A simulator to analyze compute node performance and energy usage that overcomes the scalability limitations of cycle accurate simulators
- A performance prediction framework that both incorporates feedback from performance counters and energy measurements
- An approach to automatically detect and prioritize performance bottlenecks with differential analysis
- An evaluation of tool support for the task concept of the OpenMP shared memory programming paradigm
- Advanced functionality in an online performance analysis framework that includes an integration with automatic performance tuning
- An approach for holistic I/O analysis that combines application and system events
- An approach addressing a specific class of memory usage errors in GPGPU accelerated applications using CUDA

These topics highlight how tools face several of the challenges that increased hardware complexities impose. At the same time, the workshop presentations include updates and demonstrations of multiple widely available tools, as well as success stories on their use. Thus, the workshop serves both as a forum for application developers that want to apply tools, as well as for tool developers as a knowledge exchange. This audience combination enables fruitful and engaged discussions of current and future challenges in order to ensure that tools meet their requirements on current and future HPC systems.

Dresden, Germany
2014

Andreas Knüpfer
José Gracia
Wolfgang E. Nagel
Michael M. Resch

Contents

1 Performance Analytics: Understanding Parallel Applications Using Cluster and Sequence Analysis 1
Juan Gonzalez, Judit Gimenez, and Jesus Labarta

2 Tools for Simulation and Benchmark Generation at Exascale 19
Mahesh Lagadapati, Frank Mueller, and Christian Engelmann

3 Suitability of Performance Tools for OpenMP Task-Parallel Programs 25
Dirk Schmidl, Christian Terboven, Dieter an Mey, and Matthias S. Müller

4 Recent Advances in Periscope for Performance Analysis and Tuning 39
Yury Oleynik, Robert Mijaković, Isaías A. Comprés Ureña, Michael Firbach, and Michael Gerndt

5 MuMMI: Multiple Metrics Modeling Infrastructure 53
Xingfu Wu, Valerie Taylor, Charles Lively, Hung-Ching Chang, Bo Li, Kirk Cameron, Dan Terpstra, and Shirley Moore

6 Cudagrind: Memory-Usage Checking for CUDA 67
Thomas M. Baumann and José Gracia

7 Node Performance and Energy Analysis with the Sniper Multi-core Simulator 79
Trevor E. Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout

8 A Comparison of Trace Compression Methods for Massively Parallel Applications in Context of the SIOX Project ... 91
Alvaro Aguilera, Holger Mickler, Julian Kunkel, Michaela Zimmer, Marc Wiedemann, and Ralph Müller-Pfefferkorn

9 PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis 107
Zakaria Bendifallah, William Jalby, José Noudohouenou, Emmanuel Oseret, Vincent Palomares, and Andres Charif Rubial

Chapter 1
Performance Analytics: Understanding Parallel Applications Using Cluster and Sequence Analysis

Juan Gonzalez, Judit Gimenez, and Jesus Labarta

Abstract Due to the increasing complexity of High Performance Computing (HPC) systems and applications it is necessary to maximize the insight of the performance data extracted from an application execution. This is the mission of the Performance Analytics field. In this chapter, we present three different contributions to this field. First, we demonstrate how it is possible to capture the computation structure of parallel applications at fine grain by using density-based clustering algorithms. Second, we introduce the use of multiple sequence alignment algorithms to asses the quality of a computation structure provided by the cluster analysis. Third, we propose a new clustering algorithm to maximize the quality of the computation structure detected minimizing the user intervention. To demonstrate the usefulness of the different techniques, we also present three use cases.

1.1 Introduction

Nowadays, High Performance Computing (HPC) systems provide the scientific community with hundreds of thousands or even millions of computing cores to run complex parallel applications. In terms of analysis, complex applications imply complex analysis work due to the huge amount of elements involved and the volume of data to process. The Performance Analytics field provides developers and analysts adapted techniques and methods from the Data Analytics area, giving a valuable insight to understand how their applications behave and perform. In this chapter, we introduce three main contributions to this field developed at the Barcelona Supercomputing Center Performance Tools Team in the last few years.

The first contribution is the use of cluster analysis to detect the fine-grain computation structure of a parallel application. Cluster analysis has been used to reduce the analysis data by grouping processes that behave similarly [1, 5, 7]. Our

J. Gonzalez (✉) • J. Gimenez • J. Labarta
Barcelona Supercomputing Center/Polytechnic University of Catalonia c/Jordi Girona, 31. 08034 Barcelona, Catalunya, Spain
e-mail: juan.gonzalez@bsc.es

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_1

proposal is to group the behaviour of computation regions defined below the process granularity, an approach shared with SimPoint [9]. We demonstrate that, at this granularity level, the density-base clustering algorithm DBSCAN [3] provides better results than K-means [4], the algorithm typically used in previous works.

The vast majority of applications follow the Single Process Multiple Data (SPMD) paradigm. The second contribution we detail is the use of Multiple Sequence Alignment (MSA) algorithms to evaluate if the groups of computation regions detected, i.e. the computation phases, reflect the SPMD structure. In bioinformatics, MSAs are widely used to evaluate similarities between DNA chains or proteins (sequences of nucleic acids or amino acids, respectively). In our context, the sequences of computations performed by the different processes in a SPMD application should be similar. The Cluster Sequence Score we propose measures this similarity, the *SPMDiness* of the application, using an MSA.

The third contribution is a new density-based clustering algorithm, the Aggregative Cluster Refinement algorithm. This algorithm overcomes some problems detected on DBSCAN, such as the sensitivity to its parametrization and the inability to detect clusters with different densities. The Aggregative Cluster Refinement iteratively refines the results obtained by DBSCAN using the Cluster Sequence Score to evaluate the clusters detected on successive steps. In addition, it generates the hierarchical description of the clusters formation, useful to understand how the different computation phases behave below the SPMD level.

To demonstrate the usefulness of the three previous contributions, we put in practice the computation structure detection in three different uses cases. The first one is a methodology to extrapolate performance data combining the computation structure detection with a multiplexing scheme to access the performance counters. The second use case is the application of the structure detected so as to determine which are the computation regions to be simulated in a multi-scale simulation framework. The third use case is a quantitative study of the potential gain when introducing software or hardware improvements that impact not all the execution but specific computations in the application, defined by the structure detection.

The rest of this chapter is organized as follows: Sect. 1.2 describes the computation structure detection using cluster analysis; the *SPMDiness* evaluation of the computation structure using MSAs is detailed in Sect. 1.3; Sect. 1.4 introduces the Aggregative Cluster Refinement algorithm; Sect. 1.5 contains the three different use cases introduced; finally, Sect. 1.6 presents the conclusions of our current work as well as opportunities for future research.

1.2 Computation Structure Detection Using Cluster Analysis

The target of the technique we present in this section is to group the behaviour of different CPU bursts executed by a parallel application. A CPU burst is the region in a parallel application between an exit to the parallel runtime and the following entry.

```
for (int i = 0; i < 50; i++)
{
  Receive(Partner)
  f1();
  if (condition)
    f2();
  else
    f3();
  Send(Partner)
}
```

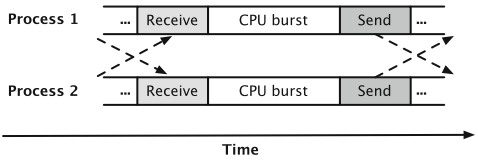


Fig. 1.1 Visual definition of a **CPU burst** based on the pseudo-code of a simple SPMD message-passing application and a single iteration timeline when executed the code with two processes. The CPU bursts cover the `f1` sub-routine and, depending on the *condition* value, `f2` or `f3` sub-routines

Figure 1.1 illustrates an example of the pseudo-code of a SPMD application and the corresponding CPU bursts in a timeline representation when executing the code with two processes. CPU bursts reflect the dynamic application behaviour observed during the execution, e.g. alternate execution paths, as opposite to the static code structure. We will discuss this later in this Section.

Each CPU burst is characterized with a vector of performance hardware counters, erasing the time and space (process) components. Current processors offer a wide variety of performance counter metrics. To select the counters (or metrics derived from the counters) used by the clustering algorithm, we follow the approach proposed by Joshi et. al [6]: to choose those counters with a clear physical interpretation by the analyst. In the experiments we present, we mainly used the metrics of "Completed Instructions", as a measure of the computational complexity of each burst, and "Instructions per Cycle" (IPC, derived from Completed Instructions and Total Cycles), as a measure of the achieved performance.

As usual in any data analysis process, a first pre-process of the data improves the analysis both in quality and execution time. In our case, we filter those bursts whose duration is less than a small threshold. This filtering avoids the analysis of those regions that do no represent effective computation, for example the bursts that appear in sequences of back-to-back MPI calls. We also apply a logarithmic normalization to the metrics whose dynamic range is wide, such as Completed Instructions, and a range normalization to all metrics, to guarantee that all of them have the same weight in the clustering process.

Finally, we apply the DBSCAN [3] algorithm, a density-based cluster algorithm, to the pre-processed data. The DBSCAN algorithm requires two parameters, *Epsilon* (or *Eps* for short) and *MinPoints*. The resulting clusters are the different subsets of the data space where distances between points are less or equal to *Eps* with more than *MinPoints* individuals. The points that do not fall into any cluster are marked as noise.

The choice of DBSCAN responds to the fact that performance counters data can have an arbitrary shape. Clustering algorithms such as K-means, used in previous works [1,5,7,9], are not able to capture the desired groups due to certain assumptions about the data distribution that are no longer true in our data. Figure 1.2 compares the clusters obtained with X-means [8] (a variant of K-means that automatically

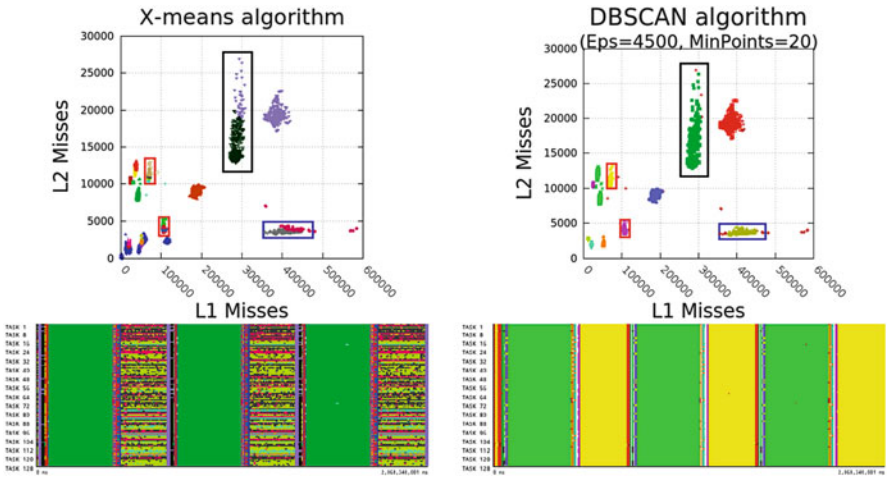


Fig. 1.2 Detected clusters and computation structure timelines using X-means algorithm (*left*) and DBSCAN (*right*)

guesses the number of clusters) and the ones obtained with DBSCAN. The data used corresponds to three iterations of an execution of CPMD (Car-Parinello Molecular Dynamics) with 128 processes.

The scatter plots axes represent the metrics used to apply the cluster analysis. These plots display clouds of points with different shapes: spherical and elliptical clusters with a strong horizontal or vertical component. In this example, we used L1 and L2 data cache misses for illustrative purposes, but other metrics also present similar shapes. In the example, X-means (left plot) tends to partition the ellipses, e.g. the cloud inside the black box was divided into two clusters. In fact, the uppermost points of this cloud were assigned to the same cluster as the cloud on the right. Applying DBSCAN (right plot), the cloud within the black box is detected as a single cluster. The blue box (on the bottom-right area of both plots) highlights a cloud with a strong horizontal component where, again, X-means detected two clusters while DBSCAN detects just one.

In terms of the computation structure, the data partition generated by DBSCAN is better than the X-means cluster assignment. This can be seen in the timelines at the bottom of Fig. 1.2. In these timelines, the X axis represents the time, the Y axis represents the 128 processes involved in the application execution, and the colour identifies the cluster of each CPU burst. In the left timeline, the clusters detected by X-means reflect a pattern that repeats three times, capturing the three iterations of the application. On each iteration, we see an initial SPMD phase in dark green, where all process are performing the same computations. On the other hand, the second phase of each iteration does not present the regular SPMD structure expected. The bursts contained in this phase are the ones marked with red boxes in the left plot, where X-means divided small compact clouds into different clusters.

DBSCAN detects these small clouds as single clusters (red boxes on the right plot). This partition of the data leads to a clear detection of the different SPMD computation phases as can be seen on the right timeline.

1.2.1 Application's Syntactic Structure and Behaviour Structure

The application syntactic structure is the structure of the application's source code. It represents the static division in different sub-routines, loops or lines the developer decided while coding the algorithms of the application. Many performance tools use this view to report their results. In particular, profiling tools present basic statistics for each component of the syntactic structure, e.g. average duration of each sub-routine. With the structure detection proposed, we detect the behaviour structure of the application. The behaviour structure captures the variability in time and space (i.e. between processes) of the performance metrics, achieved by the different sections of the program. These two structures are complementary, as they provide two different points of view in the analysis process. The following two examples demonstrate that there is not a bijective relationship between the syntactic structure and the behaviour structure.

In the first example we show how different routines of the syntactic structure present a common behaviour in the behaviour structure. This is the case of three main sub-routines of the NPB BT benchmark (`x_solve`, `y_solve` and `z_solve`). The top timeline in Fig. 1.3 shows how these sub-routines are distributed along four iterations of the main application loop (plus the minor routine `copy_face`). The bottom timeline presents the computation phases detected by our cluster analysis. Comparing both timelines, one can see that sub-routine `copy_faces` corresponds unequivocally to Cluster 4, the `y_solve` and `z_solve` routines present exactly the same behaviour (Cluster 3 plus Cluster 1 phases), and `x_solve` has two phases: the first detected as Cluster 2 and the latter detected as Cluster 1. This common behaviour corresponds to the actual semantics of the algorithm, which applies the same solvers (with the same behaviour) on each dimension of a three-dimensional space.

The second example shows that a single sub-routine of the Hydro Solver exhibits two different behaviours. In Fig. 1.4, the top timeline depicts the application sub-routines. We have marked two executions of the same sub-routine, `update-ConservativeVars`. Observing the behaviour structure detected in the bottom timeline, we can see that the first execution of this sub-routine was detected as Cluster 11 (dark grey) and second execution as Cluster 10 (olive), indicating a multi-modal behaviour of this routine. The scatter-plot on the right shows that the main difference between both clusters is the total number of instructions executed. We observed in the application source code that in the first invocation of the routine, it traverses a 2D data mesh by rows, and in the second invocation, by columns. This

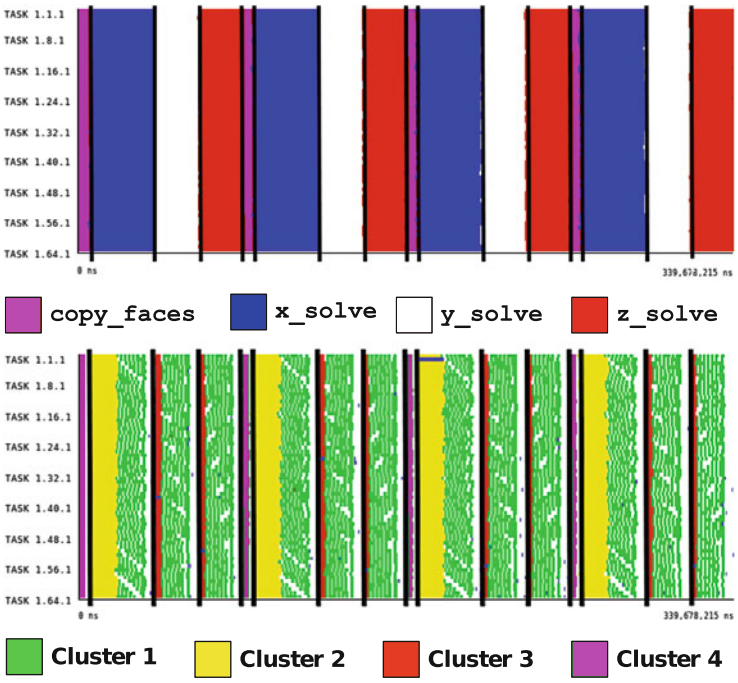


Fig. 1.3 Comparison of the syntactic structure of the NPB BT benchmark (*top*) an the behaviour structure detected with DBSCAN (*bottom*)

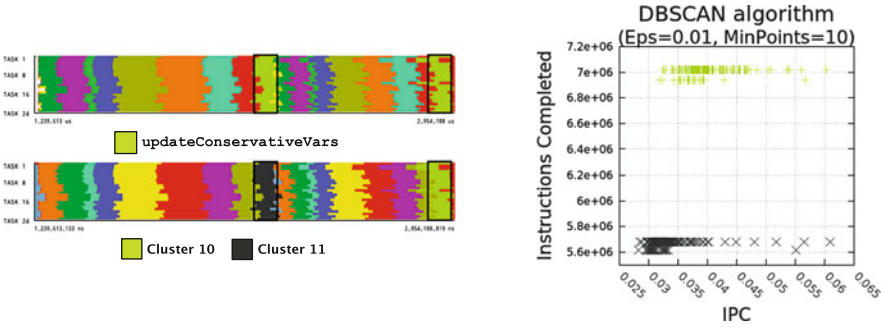


Fig. 1.4 Comparison of the Hydro Godunov Solver sub-routines (*top timeline*) the computation structure detected using DBSCAN (*bottom timeline*) and plot detail of the Clusters 10 and 11

different data access patterns take different execution paths in the routine, resulting in a different number of instructions executed.

1.3 Sequence Analysis

As stated before, the vast majority of parallel applications follow the SPMD paradigm. When using DBSCAN, the parameter selection plays a crucial role to capture this structure. The plot and timeline on the left of Fig. 1.5 show the results of clustering a single iteration of the NPB BT benchmark with fixed *MinPoints* = 10 and low *Eps*, resulting in high density clusters. This *restrictive* value of *Eps* leads to a very fine detection of the computation structure showing an initial SPMD phase but patterns below the SPMD structure in the rest of the execution. The plot and timeline on the right show the results obtained with the same data, same *MinPoints* parameter but a higher *Eps* value, that clearly reveals the SPMD structure. Both structures are correct, but the second approach makes the tasks of the analysis easier focusing on clear phases of the application. This claim constitutes our *expert criterion* to asses the quality of a given structure: it must show a series of fine-grain SPMD phases where all processes of the application execute the same cluster at the same time. The Cluster Sequence Score is the technique we propose to apply this criterion automatically, measuring what we call the *SPMDiness* of an application computation structure.

To implement the Cluster Sequence Score we made the following observation: if in a SPMD application all processes perform the same computations at the same time, the sequence of computations in all processes should be the same. Consequently, the sequence of clusters detected should also be the same for all processes. To compare these sequences we use a Multiple Sequence Alignment

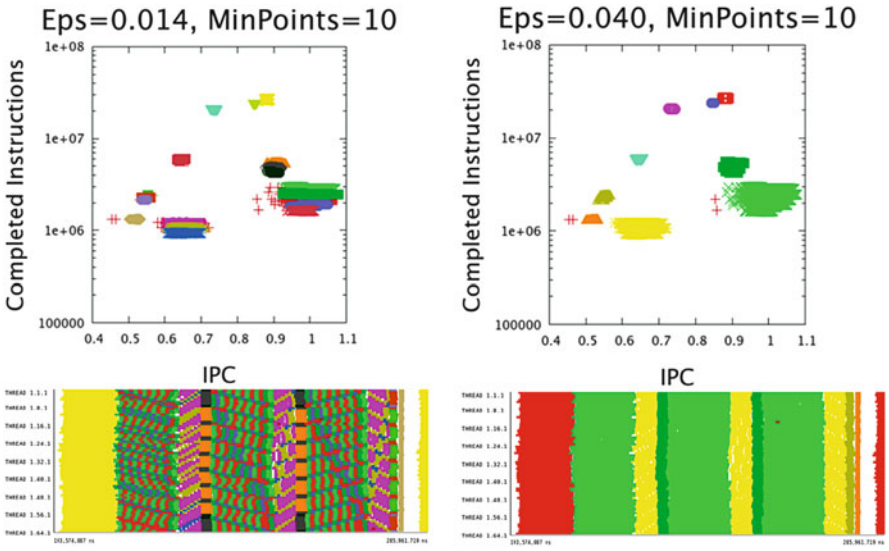


Fig. 1.5 Comparison of the structure detection results using DBSCAN with the same value of *MinPoints* and a low *Eps* value, on the *left hand*, and higher one, on the *right hand*

(MSA) algorithm, typically used in bioinformatics, for example to check similarities between proteins. The analogy is simple: each cluster is represented as an amino-acid, each process is represented as a protein, and the whole application is the set of proteins to compare.

The output of the MSA algorithm is a matrix of modified sequences with possible gaps to maximize the similarities between them. Traversing this matrix column by column we compute all the *Scores per Cluster*, the average percentage of the total number of tasks that execute that cluster on each position in the sequence where the cluster appears, i.e. how well it represents a SPMD phase. Finally, we compute the `Global Score`, a sum of all the *Scores per Cluster* weighted by their aggregated duration. The *Global Score* determines the *SPMDiness* of a given application structure.

1.4 Computation Structure Refinement

In addition to the difficulty on selecting the appropriate DBSCAN parameters discussed in the previous section, we also observed an important issue with this algorithm: it fails to detect the correct clusters when the density varies across the data space. This is caused by the use of a single *Eps* parameter to define the neighbourhood searches.

Figure 1.6 shows an example of this issue. The left plot and timeline show the results of using a high *Eps* value. With this parameter, the algorithm distinguished correctly four clusters with high density inside the black box. Those clusters correspond to four clear SPMD phases located at the second half of the timeline.

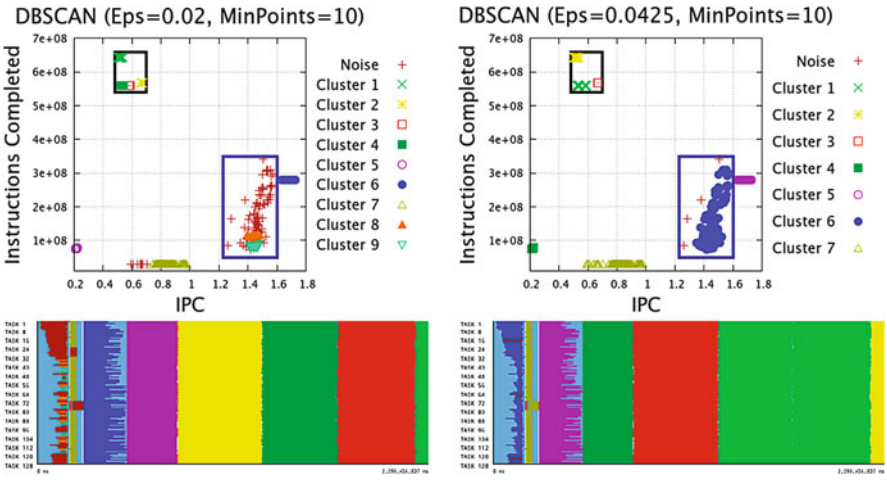


Fig. 1.6 Resulting clusters and phases of two executions of DBSCAN for the same application

Contrarily, the algorithm did not detect as a single cluster the cloud points inside the blue box, which actually correspond to a single SPMD phase with high variability in terms of duration, located at the initial section of the timeline, depicted in brown.

Increasing the value of *Eps* (maintaining the same value for *MinPoints*) also leads to an improper detection of SPMD phases. On the right plot, the points inside the blue box are now detected as a single cluster, representing the desired SPMD phase at the initial part of the right timeline. On the other hand, two of the clusters inside the black box have merged into a single one, which in terms of the computation structure implies a loss of detail, as two distinct SPMD phases are detected as a single one. This effect is known as *over-aggregation*.

1.4.1 Aggregative Cluster Refinement

The Aggregative Cluster Refinement addresses the problems observed in DBSCAN. It is an iterative algorithm that executes DBSCAN n times (n is the only parameter to the algorithm), with a fixed value of *MinPoints* and a different *Eps* on each iteration to detect clusters with different densities. The *Eps* values are generated traversing the *k-neighbour distance* plot described in [3] and sorted increasingly to search for dense clusters first, which is computationally cheaper. After running DBSCAN, the clusters are evaluated using a quality score (in our case, the Cluster Sequence Score described in Sect. 1.3). Those clusters whose *Score per Cluster* is higher than a given value do not take part in further iterations, reducing the volume of data as the algorithm progresses. The algorithm converges when all clusters represent SPMD regions or when there are no more *Eps* values to explore. This algorithmic scheme is equivalent to the one used by X-means to refine the clusters obtained using K-means.

In addition to the iterative clusters formation, the Aggregative Cluster Refinement builds the *refinement tree*, which depicts the hierarchical relationship between the clusters obtained using different *Eps* values. This tree is similar to the dendrogram produced by the agglomerative hierarchical clustering methods, and provides a view of the application structure below SPMD.

Figure 1.7 shows the results of applying the Aggregative Cluster Refinement to the same data set of Fig. 1.6. The algorithm was executed with parameter $n = 10$. On the left, each level of the refinement tree corresponds to one iteration of the algorithm where the nodes represent the clusters detected by DBSCAN using the selected *Eps* for that level (low to high values from top to bottom); the filled nodes mark those clusters that represent SPMD phases; the empty nodes represent those clusters that still do not capture the desired SPMD phases and require further refinement; the values inside the nodes indicate the cluster identifier and its *Score per Cluster*.

Comparing the refinement tree and clusters in the scatter plot on the right, we can observe that in the first level the algorithm detected the Clusters 1, 2, 3 and 4. These clusters have high density and appear as the compact clouds in the upper-right region of the plot. Clusters 4 and 5 appear at level 3 and 4 respectively, which

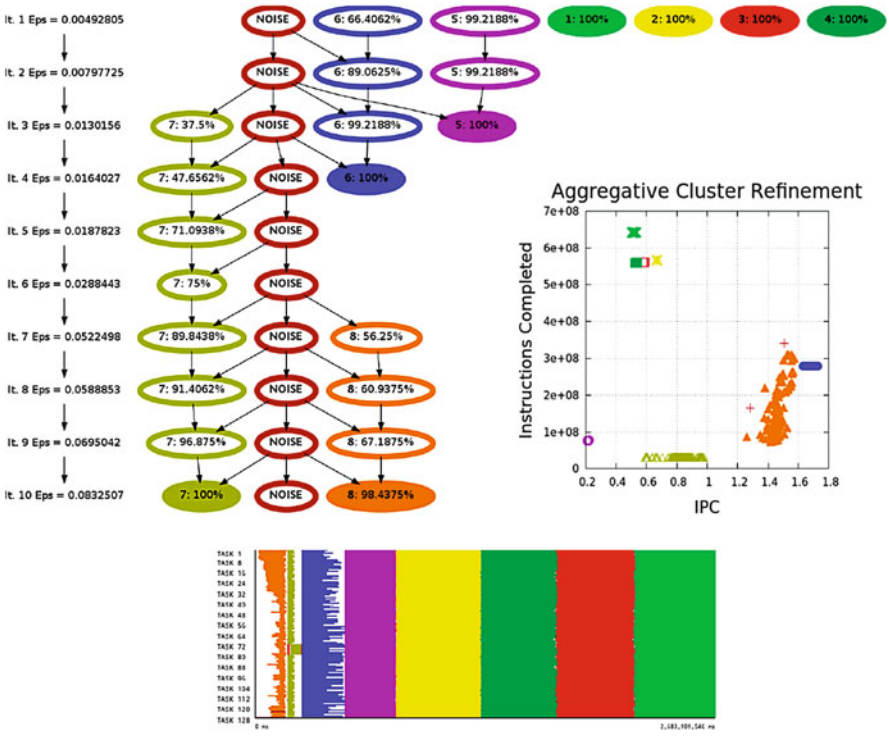


Fig. 1.7 Results of applying the Aggregative Cluster Refinement algorithm to the Hydro Solver input data also used in Fig. 1.6

imply that are less dense and have more variability. Finally, the algorithm detected the least dense Clusters 7 and 8 at level 10. These cluster are the sparse clouds at the center-bottom and right-bottom regions of the plot. Looking at the timeline at the bottom of Fig. 1.7 it is easy to verify that the clusters detected perfectly represent the SPMD structure of the application at fine grain, surpassing the results obtained by DBSCAN and just requiring the number of *Eps* to explore, which is far more understandable to the user than selecting *Eps* and *MinPoints*.

1.5 Uses Cases

In this Section we illustrate three different scenarios where the computation structure detection based on the Aggregative Cluster Refinement is used to aid in the performance analysis process.

1.5.1 Performance Data Extrapolation

Performance hardware counters provide an invaluable information for the performance analysis task. However, as discussed by Sprunt in [10], there are always limitations on the amount of counters that can be read simultaneously and the combinations of different counters. For example, the available registers in current processors for performance accounting typically range from 4 to 8, and the way to access them is in fixed groups of counters defined during the processor design. We present in this section an extrapolation methodology to characterize the different computation phases detected using cluster analysis with more performance counters than the ones that can be read simultaneously, with minimum error and avoiding multiple executions of the application.

This methodology is based on combining the structure detection and a multiplexed acquisition of the performance counters data. Figure 1.8 shows four different elements that intervene in the extrapolation process. The upper timeline depicts the time multiplexing data acquisition, where the processes shift between two counters groups every certain time (other multiplexing schemes are also applicable). The second timeline depicts the computation structure detected using the cluster

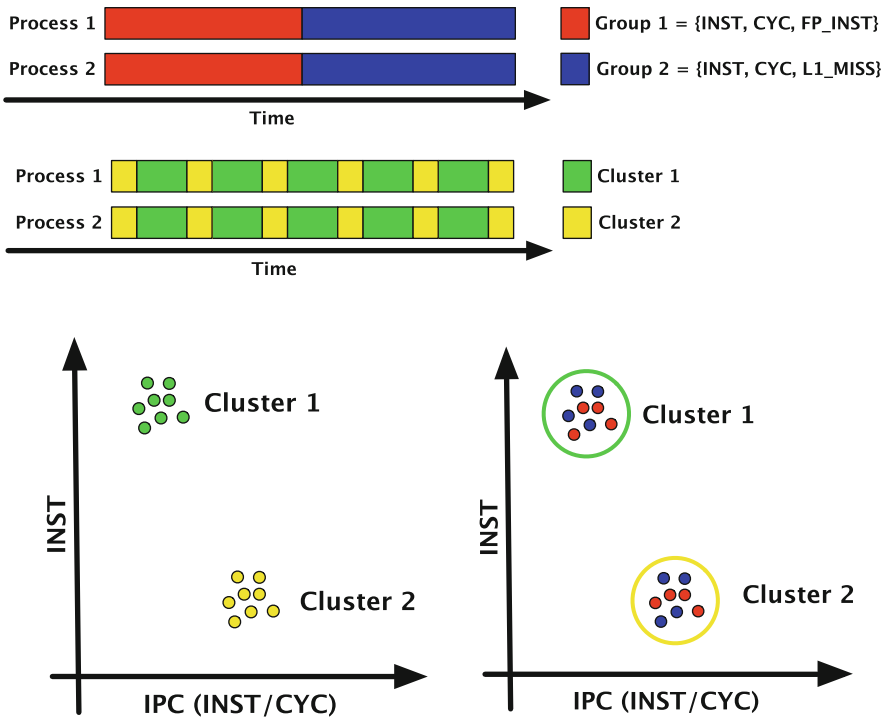


Fig. 1.8 Scheme of the performance data extrapolation methodology

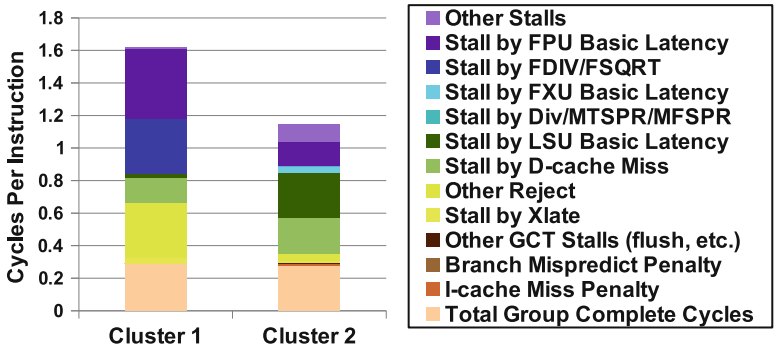


Fig. 1.9 CPI breakdown model of two computation phases detected in PEPC computed from a single execution of the application using the performance extrapolation methodology

analysis. To perform this cluster analysis it is required that all counters groups share a common subset of counters. In all processors we used in our experiments we found that Completed Instructions and Total Cycles are always available in all counters groups. Next, the bottom left plot shows the detected clusters using the counters common to all groups. Finally, the bottom right plot is the key to understand the methodology: as we have multiplexed the counters groups, different CPU bursts of each cluster have measurements from the different counters groups. Combining the data from the different bursts, we compute the average value of all counters present in all groups for all the clusters detected.

Figure 1.9 shows the CPI breakdown model defined in [11] of the two main phases of PEPC application, a plasma physics parallel-tree code. To compute this model, it is necessary to read six different counters groups. Using the extrapolation methodology we can compute it with a single execution of the application, saving five extra executions.

1.5.2 Multi-scale Simulation Framework

The multi-scale simulation methodology we propose consists in combining two simulators to predict the behaviour of message-passing parallel applications. We use an application-level simulator, Dimemas, to predict the network behaviour of the application and a micro-architecture simulator, MPSim, to predict the behaviour of the application computation regions. This combination of simulators is similar to the approach presented in [2].

In Fig. 1.10 you can see the scheme of our multi-scale simulation methodology. Starting with a trace of a parallel application (step 1), we produce a sub-trace (or trace cut) containing information of just two iterations (step 2). A cluster analysis is applied to the information of the computation regions present on this reduced

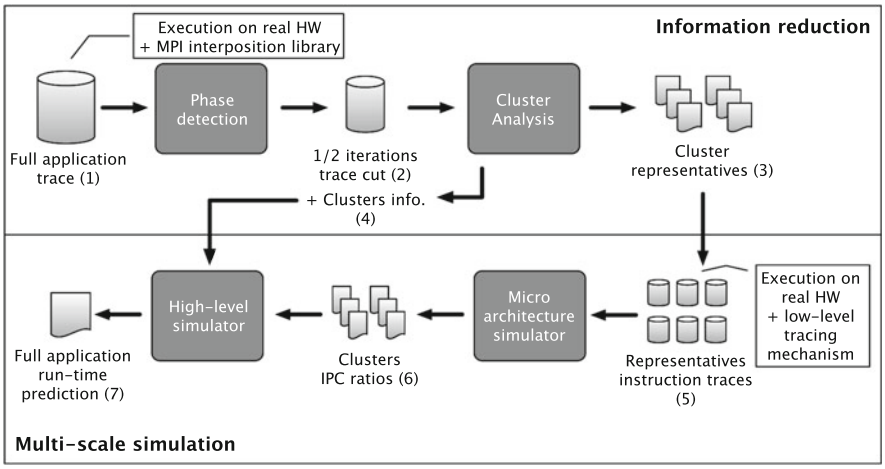


Fig. 1.10 Scheme of the multi-scale simulation framework

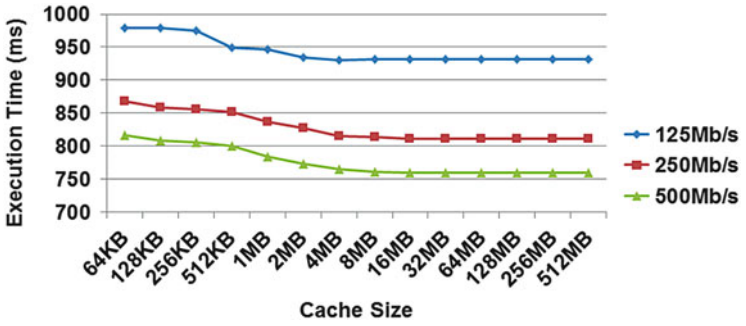


Fig. 1.11 Predicted execution times of the NPB using the multi-scale simulation methodology

trace, and a set of representatives per cluster is selected (step 3), adding cluster information to the trace cut (step 4). The set of representatives is traced (step 5) and simulated using a low-level simulator to obtain the ratios on other possible processor configurations (step 6). Finally, using a full-system-scale simulator, we combine the communication information present in step 3 and the cluster instructions per cycle (IPC) ratios (step 6) to predict the total runtime of the whole application (step 7).

The main difference between our methodology and the one presented in [2] is the addition of the process to reduce the total number of instructions to simulate in the micro-architecture simulator, the most time-consuming element of the whole framework (marked as "Information reduction" in the methodology scheme). Thanks to this reduction process, we can perform very detailed analyses in a reasonable time. For example, in Fig. 1.11 we predict the execution time of the NPB BT benchmark using networks with different bandwidths and processors with

different cache sizes. Simulating all instructions in this application would have been absolutely unfeasible (months or even years of simulation time) but it just took less than a day of to simulate these scenarios with our methodology.

1.5.3 Fine-Grain Performance Prediction

In this use case, we also apply the Dimemas simulator, combined with the computation structure detection provided by the cluster analysis to bring up new studies of the potential of software and hardware modifications that were not possible to evaluate before. To illustrate these new studies, we present the analysis of the PEPC application, previously used in the data extrapolation methodology. The application was executed using 256 MPI processes.

As a result of the cluster analysis, we detected two different computation phases, perfectly SPMD. Table 1.1 shows a a profile of different performance metric for each cluster. Cluster 1 covers 57.02 % of the total computation time and Cluster 2 the 40.98 %. The remaining 2 % corresponds to the filtered CPU bursts. We are particularly interested two of the metrics presented to define the studies of the following subsections. The first metric we focus on is the *AverageIPC*, which points that the performance of Cluster 1 is around 25 % less than Cluster 2. The second metric is the duration load balancing (LB_{dur}), computed as the division between the maximum burst duration and the average burst duration for a given cluster. It expresses, in a range between 0 (poor) and 1 (perfect), the computational balance of such a region. In this case, the values obtained indicate that there is a slight imbalance in both clusters.

1.5.3.1 Evaluation of Software Improvements

In this first study we measure, in terms of application execution time reduction, two potential software improvements without requiring any code modification. These improvements are based on the observations regarding the clusters performance highlighted previously.

The first improvement we consider is what would happen if we increased Cluster 1 IPC to 0.8 (similar to the average IPC of Cluster 2). To simulate this, we apply in

Table 1.1 PEPC clusters characterization

Metric	Cluster 1	Cluster 2
Total computation time (%)	57.02	40.98
Average burst duration (ms)	57.954	41.892
Average completed instructions	8.130×10^7	8.260×10^7
Average IPC	0.619	0.870
LB_{dur}	0.859	0.841

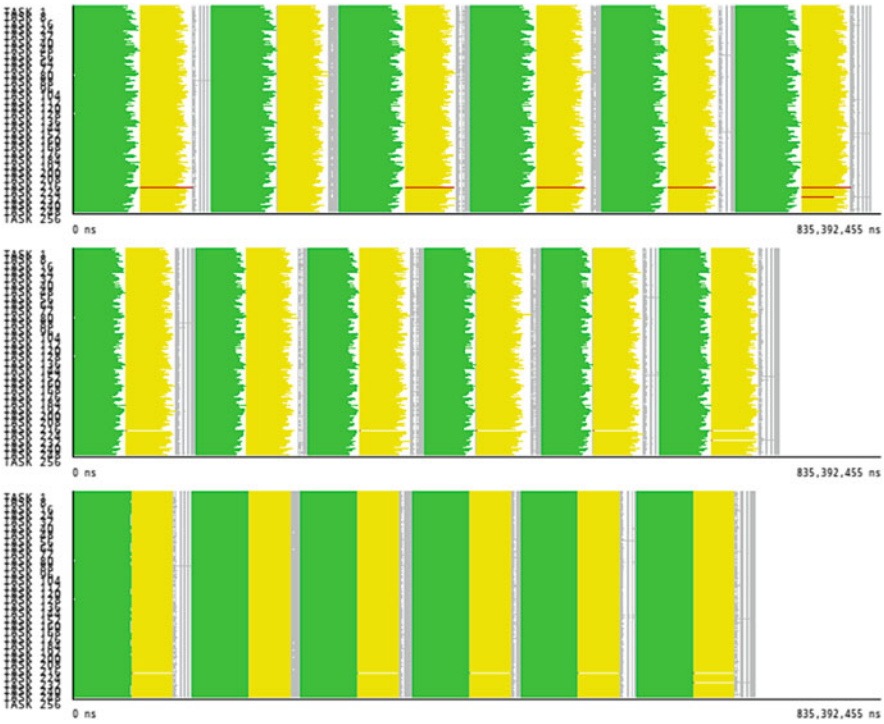


Fig. 1.12 Comparison of nominal (*top*), improved IPC of Cluster 1 (*middle*) and duration balancing (*bottom*) simulations of PEPC

Dimemas a duration scale factor to all bursts belonging to Cluster 1. This ratio is computed as the division between the average IPC obtained for such cluster and 0.8 (the target IPC). The second improvement is what would happen if we were able to perfectly balance the duration of both clusters. To perform this second simulation, we use a Dimemas feature to substitute the durations of all bursts inside a cluster by a fixed duration. In this case, we used the average burst duration listed in the Table 1.1.

Figure 1.12 shows the timelines of these simulations. Top timeline is the nominal simulation, i.e. simulation with the parameters of the original machine. It serves as a reference to quantify the reductions in the execution time. Middle timeline shows the IPC improvement simulation. Bottom timeline corresponds to the duration balance simulation. The results show that improving the IPC of Cluster 1 we will obtain a 13 % reduction of the total application execution time. This reduction grows up to a 19 % when balancing both clusters. With these figures, the application developer could take a decision about where to put the resources so as to maximize the return of the optimization efforts.

1.5.3.2 Evaluation of the Use of New Hardware

The second study correspond to a measure, in terms of speedup, of two potential changes in hardware elements in the system where the application is executed: first, to use faster general purpose CPUs, and hence all the computations speedup; second, to use hardware accelerators, where just the computations of specific clusters are improved.[1] In both cases, we consider a CPU/accelerator speedup factor in the range from 1 to 64. Additionally, we also increase the interconnection network bandwidths in a range from 256 MBps to 16 GBps. In this study, the structure detection is required when simulating the gain provided by the accelerators, to define which regions of the application will be get benefited of using this hardware.

Figure 1.13 shows two surface plots with the results of the different simulations. The left plot presents the speedups obtained with general purpose CPUs and the right plot using accelerators. Comparing both, we can observe that the maximum speedup in the accelerator (right plot) is close to 8, when using a 16 GBps network and 64× CPU ratio accelerators. The same improvement is reached when using a general purpose CPU of ratio 8, indifferently to the network bandwidth (left plot). Current CPUs are easily 8× faster than the PowerPC 970MP processor where the application was executed, but obtaining a 64× speedup with acceleration hardware can be costly, taking into account that the speedup requires not just the hardware, but also a code refactoring. In terms of the network, the application does not get any benefit from having an available bandwidth larger than 1–2 GBps in the case of general purpose CPUs, while when using accelerators there are no speedup improvements at all. As in the software improvement study, these predictions are useful to evaluate which elements on a HPC facility would be more profitable to update, without requiring the actual hardware.

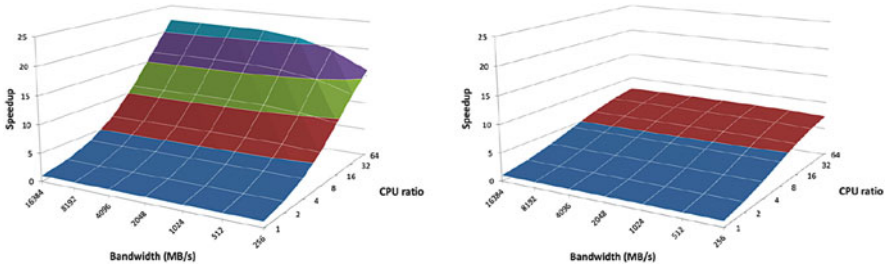


Fig. 1.13 PEPC speedups achieved by using different network bandwidths and general purpose CPUs of different speeds, *left*, and accelerators of different speeds, *right*

[1] This scenario also mimics using OpenMP to parallelize the large computation regions.

1.6 Conclusions and Future Work

In this chapter, we have presented the contributions to the Performance Analytics area developed in the last few years at the BSC Performance Tools Team. These contributions are based on cluster and sequence analysis, and face the challenge of automatically detecting parallel applications computation phases. With this knowledge, we facilitate the tasks of analysis by providing an intuitive and understandable insight about how the applications behave.

Our current research interests move forward to higher scales and to an on-line scenario, with the aim of maximizing the insight we can extract from large-scale applications during their execution while minimizing the volume of data recorded.

Acknowledgements The work presented in this chapter has been partially founded by IBM, through the IBM-BSC MareIncognito collaboration agreement, the Spanish Ministry of Education under grant BES-2005-7919 and project TIN2007-60625, and the EU/Russia joint project HOPSA.

References

1. Ahn, D.H., Vetter, J.S.: Scalable analysis techniques for microprocessor performance counter metrics. In: ACM/IEEE Conference on Supercomputing (SC), Baltimore (2002)
2. Carrington, L., Snavely, A., Gao, X., Wolter, N.: A performance prediction framework for scientific applications. In: 3rd International Conference on Computational Science (ICCS), Saint Petersburg/Melbourne (2003)
3. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: 2nd International Conference on Knowledge Discovery and Data Mining (KDD), Portland (1996)
4. Hartigan, J., Wong, M.: Algorithm AS 136: a K-means clustering algorithm. J. R. Stat. Soc. Ser. C (Appl. Stat.) **28**, 100–108 (1979)
5. Huck, K.A., Malony, A.D.: PerfExplorer: a performance data mining framework for large-scale parallel computing. In: ACM/IEEE Conference on Supercomputing (SC), Seattle (2005)
6. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.K.: Measuring benchmark similarity using inherent program characteristics. IEEE Trans. Comput. **55**(6), 769–782 (2006)
7. Nickolayev, O.Y., Roth, P.C., Reed, D.A.: Real-time statistical clustering for event trace reduction. Int. J. Supercomput. Appl. High Perform. Comput. **11**(2), 144–159 (1997)
8. Pelleg, D., Moore, A.W.: X-means: extending K-means with efficient estimation of the number of clusters. In: 17th International Conference on Machine Learning (ICML), Stanford (2000)
9. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose (2002)
10. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro. **22**(4), 64–71 (2002)
11. Vianney, D., Mericas, A., Maron, B., Chen, T., Kunkel, S., Olszewski, B.: CPI analysis on POWER5, Part 2: introducing the CPI breakdown model. http://www-128.ibm.com/developerworks/library/pa-cpipower2

Chapter 2
Tools for Simulation and Benchmark Generation at Exascale

Mahesh Lagadapati, Frank Mueller, and Christian Engelmann

Abstract The path to exascale high-performance computing (HPC) poses several challenges related to power, performance, resilience, productivity, programmability, data movement, and data management. Investigating the performance of parallel applications at scale on future architectures and the performance impact of different architecture choices is an important component of HPC hardware/software co-design. Simulations using models of future HPC systems and communication traces from applications running on existing HPC systems can offer an insight into the performance of future architectures. This work targets technology developed for scalable application tracing of communication events and memory profiles, but can be extended to other areas, such as I/O, control flow, and data flow. It further focuses on extreme-scale simulation of millions of Message Passing Interface (MPI) ranks using a lightweight parallel discrete event simulation (PDES) toolkit for performance evaluation. Instead of simply replaying a trace within a simulation, the approach is to generate a benchmark from it and to run this benchmark within a simulation using models to reflect the performance characteristics of future-generation HPC systems. This provides a number of benefits, such as eliminating the data intensive trace replay and enabling simulations at different scales. The presented work utilizes the ScalaTrace tool to generate scalable trace files, the ScalaBenchGen tool to generate the benchmark, and the xSim tool to run the benchmark within a simulation.

2.1 Introduction

This decade is projected to usher in the period of exascale computing with the advent of systems of up to one billion tasks and possibly as many cores. Scaling applications to such levels poses significant challenges that cannot be met with

M. Lagadapati • F. Mueller
Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, USA
e-mail: mueller@cs.ncsu.edu

C. Engelmann (✉)
Oak Ridge National Laboratory, Oak Ridge, TN, USA
e-mail: engelmannc@ornl.gov

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_2

current hardware technologies. To assess the requirements for exascale hardware platforms and to gauge the potential of novel technologies, hardware simulation plays an important role in exascale projections. Significant challenges exist even at the single node level, the network interconnect and at the system level when trying to orchestrate the execution of such extensive numbers of cores as projected for exascale. Hardware simulators are vital in assessing the potential of different approaches under these challenges. Yet, these simulators need to be subjected to realistic application workloads that originate in the HPC realm. Currently, no such realistic workloads derived from large-scale HPC applications exist. This reduces simulation to studies of micro-kernels and assessment of peak metrics (bandwidth/latencies) without any notion of sustained application performance.

2.2 Overall Goal

This effort is targeted at alleviating the shortcomings of current hardware simulation practice by developing a universal skeleton generation capability that accurately reflects communication workloads for large-scale HPC codes.

The objective of this work is to complement the xSim simulator from Oak Ridge National Laboratory (ORNL) with benchmark generation capabilities.

To this end, the following approach has been taken:

1. ScalaBenchGen from North Carolina State University (NCSU) has been extended to auto-generate source code suitable for evaluation under xSim.
2. We have combined the ScalaBenchGen and xSim capabilities for sample HPC benchmarks/applications.

2.3 Our Prior Work and Related Work

Our work builds on ScalaTrace, an MPI tracing toolkit with aggressive and scalable trace compression. ScalaTrace's compression can result in trace file sizes orders of magnitude smaller than previous approaches or, in some cases, even near constant size regardless of the number of nodes or application run time [4].

ScalaTrace collects communication traces using the profiling layer of MPI (PMPI) [1] through Umpire [7] to intercept MPI calls during application execution. On each node, profiling wrappers trace all MPI functions, recording their call parameters, such as source and destination of communications, but without recording the actual message content.

ScalaTrace performs two types of compression: $intra-node$ and $inter-node$. For the intra node compression, the repetitive nature of timestep simulation in parallel scientific applications is used. Intra-node compression is performed on-the-fly within a node. Further, the inter-node merge exploits the homogeneity

in behavior across different processes running the application due to the HPC-prevalent single-program-multiple-data (SPMD) programming style. Inter-node compression is performed across nodes by forming a radix tree structure among all nodes and sending all intra-node compressed traces to respective parents in the radix tree. At the parent, the respective trace representations are merged, reduced and then compressed exploiting domain-specific properties of MPI. Once propagated to the root of the radix tree, this results in a single compressed trace file capturing the entire application execution across all nodes. The compression algorithms are discussed in detail in other papers [5, 6].

As a result of these techniques, ScalaTrace produces near constant size traces by applying pattern based compression. It uses extended regular section descriptors (RSD) to record the participating nodes and parameter values of multiple calls to a single MPI routine in the source code across loop iterations and nodes in a compressed manner [2]. Power-RSDs (PRSD) recursively specify RSDs nested in a loop [3].

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation duration between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation). ScalaTrace records histograms of delta times for each instance of a particular computation, i.e., distinguishing disjoint call paths by separate histogram instances.

We also developed ScalaExtrap, a trace extrapolation tool [9]. It contributes a set of algorithms and techniques to extrapolate a trace of a large-scale execution of an application from traces of several smaller runs. We further developed a probabilistic replay capability based on approximate matching of communication events and parameters across nodes [11]. This technique reduces trace sizes for non-SPMD codes where lossless compression techniques fail. For large-scale applications with non-SPMD or ARM-based communication patterns, such techniques could also be employed for single-node replay in the future. Another interesting direction would be to assess if receiver message content can also be replayed in a probabilistic manner for a subset of messages and, if so, how to automatically identify such messages.

Most relevant to this project is the ScalaBenchGen work [8]. It contributes an automated approach to the creation of communication benchmarks. Given an MPI application, we utilize ScalaTrace to obtain a single trace file of an HPC application run that reflects the behavior of all nodes. The trace subsequently expanded to C source code by a novel code generator. This resulting benchmark code is compact, portable, human-readable, and accurately reflects the original application's communication characteristics and runtime characteristics. Experimental results demonstrate that generated source code of benchmarks preserves both the communication patterns and the wallclock time behavior of the original application. Such

automatically generated benchmarks not only shorten the transition from application development to benchmark extraction but also facilitate code obfuscation, which is essential for benchmark extraction from commercial and restricted applications.

2.4 Design and Implementation

We have complemented ORNL's xSim simulator with benchmark generation capabilities. To this end, ScalaBenchGen from NCSU was extended to auto-generate source code suitable for evaluation under xSim. The xSim simulator already has ample network topology support. ScalaBenchGen complements these capabilities with the ability to extract communication benchmark skeletons from actual HPC runs of applications. These skeletons include timings for computational parts and actual MPI communication calls. We have combined the ScalaBenchGen and xSim capabilities for sample HPC benchmarks/applications. Timings for the computational part have been enhanced to allow adaptation with respect to future (exascale) architectures. This co-design exploration supports the research path toward exascale.

Our initial version of ScalaBenchGen [8] is based on the first version of ScalaTrace [5] which produces lossless constant size traces for Single Program, Multiple Data (SPMD) parallel applications. ScalaTrace-2 [10] enhances the base version to achieve better compression for Multiple Program, Multiple Data (MPMD) parallel applications. ScalaTrace-2 is redesigned in every aspect such that data elements in trace are elastic and self explanatory. Because of these changes ScalaBenchGen is incompatible with the trace format produced by ScalaTrace-2. Hence, we have redesigned the ScalaBenchGen tool to generate benchmarks from traces of ScalaTrace-2.

As shown in Fig. 2.1, the application is linked with the ScalaTrace library to produce a trace file. The benchmark generator takes this trace file as an input and outputs the benchmark program. The benchmark generator can be run on a standalone machine. For every event present in trace, corresponding MPI event code is generated. Each event in the trace is reflected with its parameters and also the time elapsed between current and previous events. The benchmark generator introduces a sleep for the corresponding delta time before the event. This allows the wall clock time of the benchmark program to closely resemble that of the original application. Generated benchmarks can be combined with xSim, ORNL's extreme scale network interconnect simulator, for evaluation.

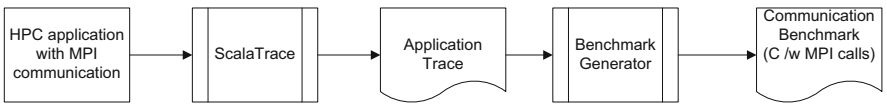


Fig. 2.1 Benchmark generation workflow

2.5 Early Results

ARC, a cluster with 1,728 cores on 108 compute nodes, 32 GB memory per node and a QDR Infiniband Interconnect is used for evaluating our benchmark generator. Benchmarks are generated for the IS and FT codes of the NAS parallel benchmark suite (NPB v3.3). Generated benchmark runtimes are compared to the execution time of the original code's execution time.

Execution times of both the original application and the generated benchmarks are similar for the FT benchmark (see Fig. 2.2a). The maximum error is 10 % for FT while 30 % maximum error is observed for IS (see Fig. 2.2b). The higher error for IS is due to replacement of MPI_Alltoallv by MPI_Alltoall within the tracing framework, which allows a more concise trace representation (at the expense of accuracy). Figure 2.2c compares the execution times of the original code and generated benchmark on both ARC and a simulated ARC environment using xSim. Currently, xSim is not supporting a fat tree configuration, which is the topology of ARC. Hence, a fat tree is loosely approximated via a star topology. This could be the reason for the differences in observed execution times but is subject to further investigation, as is the evaluation of more NPB codes.

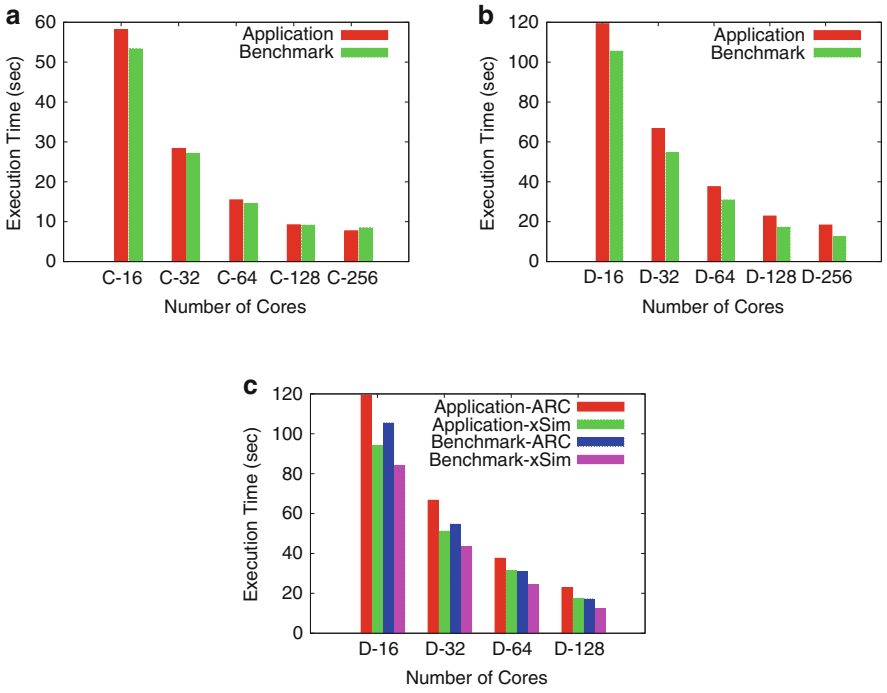


Fig. 2.2 Timing accuracy of different benchmarks. (**a**) FT (**b**) IS (**c**) IS with xSim

2.6 Conclusions

This work has demonstrated the capability to utilize ScalaTrace to generate concise and near lossless scalable communication traces to drive HPC archectural simulations. The resulting traces are transformed by ScalaBenchGen into a benchmark code. This code is subsequently fed into xSim to run the benchmark within a simulation environment. Ongoing work focuses on handling more benchmarks during the generation process and novel simulation techniques to handle exascale size workloads.

Acknowledgements This work was supported in part by NSF grants 1217748, 0937908 and 0958311, as well, as a subcontract from ORNL. Research sponsored in part by the Laboratory Directed Research and Development Program of ORNL, managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. De-AC05-00OR22725. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

1. MPI-2: Extensions to the Message Passing Interface (July 1997). http://micro.ustc.edu.cn/Linux/MPI/mpi-20.pdf
2. Havlak, P., Kennedy, K.: An implementation of interprocedural bounded regular section analysis. IEEE Trans. Parallel Distrib. Syst. **2**(3), 350–360 (1991)
3. Marathe, J., Mueller, F.: Detecting memory performance bottlenecks via binary rewriting. In: Workshop on Binary Translation (Sept 2002)
4. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalable compression and replay of communication traces in massively parallel environments. In: International Parallel and Distributed Processing Symposium, Long Beach (April 2007)
5. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: ScalaTrace: scalable compression and replay of communication traces in high performance computing. J. Parallel Distrib. Comput. **69**(8), 969–710 (2009)
6. Ratn, P., Mueller, F., de Supinski, B.R., Schulz, M.: Preserving time in large-scale communication traces. In: International Conference on Supercomputing, Island of Kos, pp. 46–55 (June 2008)
7. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with umpire. In: Supercomputing, Dallas, p. 51 (2000)
8. Wu, X., Deshpande, V., Mueller, F.: ScalaBenchGen: auto-generation of communication benchmark traces. In: International Parallel and Distributed Processing Symposium, Shanghai (April 2012)
9. Wu, X., Mueller, F.: ScalaExtrap: trace-based communication extrapolation for SPMD programs. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Antonio, pp. 113–122 (Feb 2011)
10. Wu, X., Mueller, F.: Elastic and scalable tracing and accurate replay of non-deterministic events. In: International Conference on Supercomputing, Eugene, pp. 59–68 (June 2013)
11. Wu, X., Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Probabilistic communication and i/o tracing with deterministic replay at scale. In: International Conference on Parallel Processing, Taipei, pp. 196–205 (Sept 2011)

Chapter 3
Suitability of Performance Tools for OpenMP Task-Parallel Programs

Dirk Schmidl, Christian Terboven, Dieter an Mey, and Matthias S. Müller

Abstract In 2008 task based parallelism was added to OpenMP as the major update for version 3.0. Tasks provide an easy way to express dynamic parallelism in OpenMP applications. However, achieving a good performance with OpenMP task-parallel programs is a challenging task. OpenMP runtime systems are free to schedule, interrupt and resume tasks in many different ways, thereby complicating the prediction of the program behavior by the programmer. Hence, it is important for a programmer to get support from performance tools to understand the performance characteristics of his application.

Different performance tools follow different approaches to collect this information and to present it to the programmer. Important differences are the amount of information which is gathered and stored and the amount of overhead that is introduced. We identify typical usage patterns of OpenMP tasks in application codes. Then we compare the usability of several performance tools for task-parallel applications. We concentrate our investigations on two topics, the amount and usefulness of the measured data and the overhead introduced by the performance tool.

3.1 Introduction

In recent years task based parallel programming paradigms became an alternative to classical thread based techniques for shared memory parallel programming. Task based paradigms for example are Cilk, Intel Threading Building Blocks and OmpSS. Also in OpenMP, which is the de-facto standard for thread parallel applications in HPC, tasking has been added in version 3.0 of the OpenMP

D. Schmidl (✉) • C. Terboven • D. an Mey
Chair for High Performance Computing, IT Center, RWTH Aachen University,
D - 52074 Aachen, Germany
e-mail: schmidl@itc.rwth-aachen.de; terboven@itc.rwth-aachen.de; anmey@itc.rwth-aachen.de

M.S. Müller
Chair for High Performance Computing, IT Center, RWTH Aachen University,
D - 52074 Aachen

JARA High-Performance Computing, Schinkelstraße 2, D - 52062 Aachen
e-mail: mueller@itc.rwth-aachen.de

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_3

specification as an alternative way to express parallelism. A task is a code region which can be executed independently from other tasks. Data sharing attributes are used to specify which data is `shared`, `private` or `firstprivate` for the task. The OpenMP runtime is free to execute the task immediately or to defer the execution, and the task may be scheduled later on the current or a different thread. This level of freedom in task scheduling by the runtime complicates to understand the performance behavior of an application, in particular because different OpenMP runtimes deliver highly different performance for application codes (as was shown in [13]). To better understand the performance of a program a variety of performance analysis tools exist, all using different techniques to analyze and present the behavior of an application. Shortly after tasks were introduced in OpenMP in 2008 several ideas were presented to support OpenMP tasks in performance analysis tools. Fürlinger et al. showed support for task profiling in the OpenMP profiling tool OMPP [2] and Lin et al. introduced prototypical support for tasks in the Sun Studio Performance Analyzer [8]. However, not all of these ideas have been adopted in the released product versions of these tools. Still nowadays some tools do not support tasks at all and even the tools supporting tasks differ a lot in the data measured and how it is displayed. This work analyzes the ability of performance tools to handle task-parallel programs and to investigate the usefulness of the presented data for performance optimization. Therefore we investigate task-parallel programs to identify most commonly used patterns for task creation and performance problems coming along with these patterns. Afterwards we investigate representative example applications with performance analysis tools to find out if these tools observe the mentioned performance problems. The rest of this work is structured as follows: first we shortly introduce the investigated tools in Sect. 3.2. Then in Sect. 3.3 we inspect common patterns to use tasks in parallel programs to identify representative example codes which we further investigate with all selected performance tools. Here we highlight strengths and weaknesses of all tools for the different applications before we finally draw our conclusions in Sect. 3.4.

3.2 Investigated Performance Tools

In the following sections production versions of performance analysis tools installed on our cluster are investigated with respect to their applicability for task-parallel OpenMP programs. The tools are:

3.2.1 The Intel VTune Amplifier XE

The Intel VTune Amplifier XE [5] is a sampling based performance measurement tool. It supports C/C++, Fortran, Java, C# and assembly and a variety of parallel programming paradigms for single node performance, like OpenMP, Pthreads or

Intel Threading Building Blocks. MPI or hybrid programs can be investigated, but no MPI specific data like messages sent is investigated. Instead, the tool delivers a single node profile for every MPI process. The Amplifier XE also supports measurements of hardware performance counters on Intel CPUs.

3.2.2 The Oracle Solaris Studio Performance Analyzer

The Oracle Solaris Studio Performance Analyzer [11] is also a sampling tool to investigate the performance of serial, OpenMP or PThread parallel or hybrid MPI/OpenMP applications. The tool can be used on Intel, AMD and Sparc CPUs and also supports hardware performance counters. The supported programming languages are C/C++, Fortran and Java.

3.2.3 The Score-P Measurement Infrastructure

In contrast to the previously mentioned analysis tools the Score-P measurement infrastructure [7] uses an event based technique instead of sampling to gather performance data of an application. At certain events, like the entry and exit of a function, data is measured and either directly stored in an event-trace or accumulated and finally stored in a profile, depending on the usage mode of Score-P. Score-P also allows online analysis of the data in combination with the Periscope tool [4]. Events can be instrumented in different ways, most compilers for example can automatically instrument function entry and exit events. For OpenMP pragmas the source-to-source instrumentation tool Opari [9] is used. If the data is stored in a profile it can be visualized with the Cube GUI and the trace data can be visualized with Vampir [10], as we have done it in the following sections. The gathered data can alternatively be analyzed with the TAU tool [12] or the Scalasca tool [3] can be used to automatically detect certain performance problems.

3.3 Investigating Task-Parallel Programs

To identify typical task creation patterns in applications the Barcelona OpenMP Benchmark Suite [1] (BOTS) and a set of task-parallel applications used at RWTH Aachen University are examined.

Table 3.1 shows if the codes use recursive functions to generate tasks or if the tasks are created iterative in a loop and if tasks are created nested inside of other tasks or not. Creating tasks in a recursive way is the most common task creation pattern used in 9 of 14 codes, followed by the iterative creation of tasks. Next we look at three example applications and analyze them with the performance tools.

Table 3.1 Task generation behavior of the BOTS benchmarks and several applications from RWTH Aachen University

Barcelona OpenMP Task Suite			RWTH Aachen University		
Application	Task creation	Nested tasks	Application	Task creation	Nested tasks
Alignment	Iterative	No	Sudoku	Recursive	Yes
FFT	Recursive	Yes	SparseCG	Iterative	No
Fib	Recursive	Yes	FIRE	Iterative	Yes
Floorplan	Recursive	Yes	NestedCP	Iterative	Yes
Health	Recursive	Yes	KegelSpan	Recursive	Yes
NQueens	Recursive	Yes			
Sort	Recursive	Yes			
SparseLU	Iterative	No			
Strassen	Recursive	Yes			

The applications are a Sudoku solver, a Conjugate Gradient Method implementation and the KegelSpan application code, an application developed at the Laboratory for Machine Tools and Production Engineering at RWTH Aachen University.[1]

3.3.1 Sudoku

First a very simple application, namely a task-parallel Sudoku solver, is inspected. For a given Sudoku board the solver determines all possible solutions of the Sudoku puzzle in a brute force manner. Figure 3.1 shows the initial configuration of the board used in this experiment on the left-hand side and the algorithm in used pseudo-code at the right-hand side. For every empty field the solver tries to insert every possible number. Only if the number is not yet used in the same row, column or block it creates a task to insert the number and with that then check the rest of the board, otherwise no task is created. In both cases the algorithm continues with the next number for the current field or with the next possible field. Every task which founds a valid number and is on the last empty field, stores the current solution as a valid solution for the puzzle. After all tasks have finished, all valid solutions are found. Note, even if the algorithm is fairly simple, it is hard to fully understand the runtime behavior. For example determining the number of used tasks highly depends on the initial board and even if the board is known, like the one in Fig. 3.1, it is difficult to determine this number a-priori. Since tasks are very useful for such dynamic algorithms this is a representative problem for task-parallel programs.

Figure 3.2 shows the runtime (dark-blue bars) and speedup (red curve) of the Sudoku solver for the board shown in Fig. 3.1 on a two-socket server equipped with Intel Sandy Bridge processors clocked at 2 GHz. The application achieves a best speedup of less than 5 with 32 threads.

[1] http://www.wzl.rwth-aachen.de/

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

Algorithm3.3.1: SUDOKU ()

for each *empty field f*
  for each *possible number i*
    if *i is already used in row, column or block*
      then *continue with* $i + 1$
      else
        copy the sudoku board
        insert i in f
        create a task to check the new board
        continue with $i + 1$
wait for all tasks to finish

Fig. 3.1 A 16 × 16 Sudoku board with initial entries (*left*) and our algorithm in pseudocode to solve the sudoku puzzle (*right*)

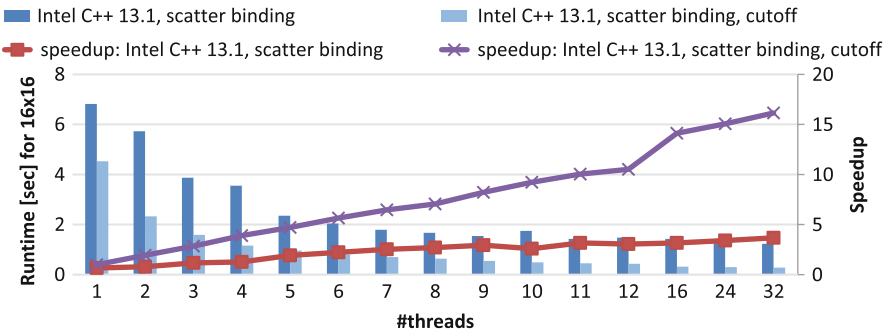


Fig. 3.2 Runtime and speedup of the Sudoku solver for the simple version and for the version with a cut-off mechanism

Obviously this is far from optimal. Next, we have a look at the performance information gathered by the investigated performance tools. With all tools we focus on execution time for our analysis. All tools allow to measure additional metrics with hardware performance counters but we are more interested how these metrics are assigned to OpenMP tasks and therefore the time metric is sufficient and by far the most important metric.

3.3.1.1 VTune

We compiled the executable with the Intel Compiler (version 13.1.1) and did several performance measurements with the Intel VTune Amplifier XE 2013 update 10. The most relevant information for the analysis of the tasking performance is shown in Fig. 3.3. First, the overview (Fig. 3.3a) presents the OpenMP tasks in line number

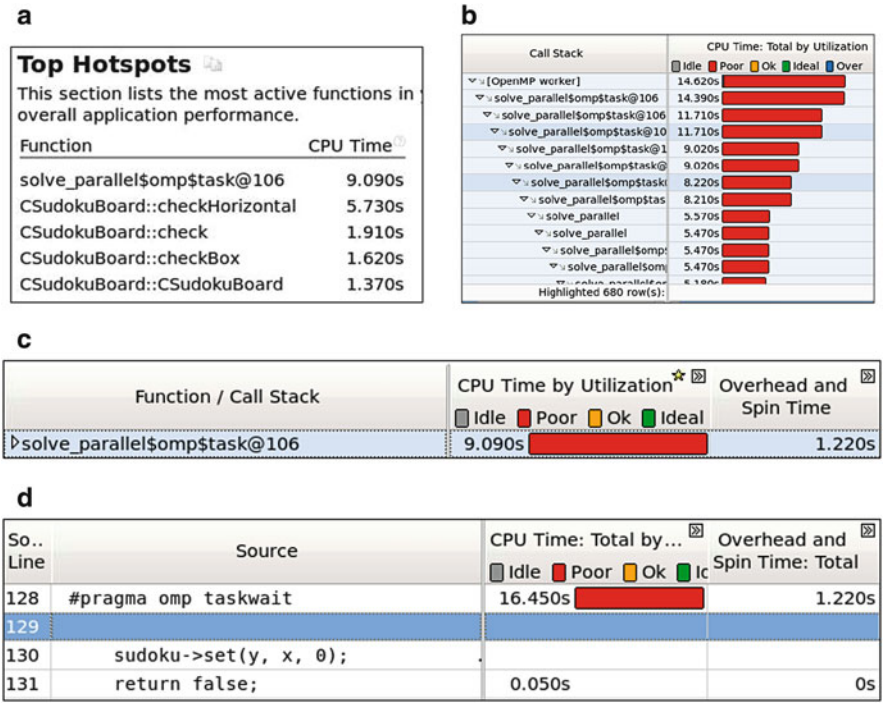


Fig. 3.3 Analysis results of the Sudoku solver with the Intel VTune Amplifier XE. (**a**) Hotspots. (**b**) Callstack. (**c**) Overhead for functions. (**d**) Overhead for sourcelines

106 as the most time consuming region in the code, the so called hotspot. Second, the callstack view (Fig. 3.3b) gives further details on this hotspot and presents that the task regions are nested recursively one inside the other. It also presents time spend on different levels of the call stack and we can observe that the time shrinks noticeably for lower levels. This shows that computation is done inside the tasks and not only in the leaf tasks at the lowest level. Third, the finest granularity is displayed at the function or source code level (Fig. 3.3c, d) where the average runtime spent on every source line can be observed, as well as a metric called "Overhead and Spin Time" which indicates time spend in the OpenMP runtime waiting or scheduling threads or tasks. For the task region 1.2 s out of 9 s are overhead, which indicates a potential performance problem.

3.3.1.2 Oracle Solaris Studio Performance Analyzer

The measurements were done with the Oracle Solaris Studio Performance Analyzer version 7.9 for Linux. The analyzed executable was compiled with the Oracle Studio 12.3 compiler for a better interoperability between the OpenMP runtime and the

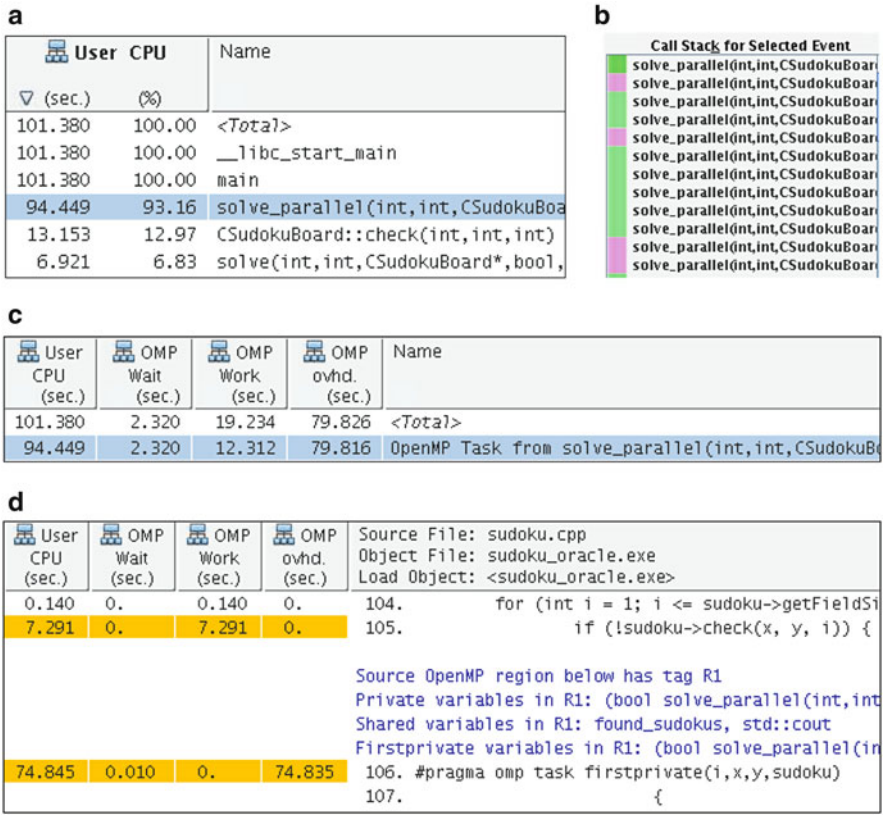


Fig. 3.4 Analysis results of the Sudoku solver with the Oracle Solaris Studio Performance Analyzer. (**a**) Hotspots. (**b**) Callstack. (**c**) Overhead for functions. (**d**) Overhead for sourcelines

performance tool. The tool delivers results similar to the Intel VTune tool. The `solve_parallel` function is identified as the hotspot (Fig. 3.4a) which is the recursive function spawning all the tasks. Also a callstack is given which indicates the recursive invocation of this function, but in contrast to VTune the Analyzer tool does not show tasks in the callstack (Fig. 3.4b), so the information that tasks are created on all these levels is not explicitly given. The time spend at each level is not presented here. Additionally to the call-stack the Analyzer presents an "OpenMP task" view (Fig. 3.4c) where the *overhead, wait and work time* for each task construct in the code is shown. Here we can see that the overhead for the task region overall is roughly 80 s, whereas the work time is only 12 s. This ratio is worse than the previous result when VTune was used, the reason is that a different OpenMP runtime (provided with the Oracle Compiler) was used which obviously incurs more overhead. Whereas the overhead of 15 % for the Intel runtime might be seen as acceptable, the overhead of 666 % here is clearly a performance problem.

The tool also gives information at the sourceline level (Fig. 3.4d), but because of the very small function we could not get any additional value from this view.

3.3.1.3 Score-P/Profiling

The Score-P measurement infrastructure (version 1.2) was used with the executable compiled by the Intel Compiler. The profiling mode was used and only OpenMP constructs were instrumented, to avoid overhead by function instrumentation. Also this tool identifies the task region in line 106 as the hotspot of the application. Figure 3.5 shows information related to task-instances which were not presented in the other tools. The direct instrumentation allows for example to identify the number of visits for the task region, meaning the number of created tasks. This is about ten million tasks in this case. Given also the overall time spent in task execution of 46 s, we can calculate an average task-instance duration of 4.6 µs. This detailed information cannot be measured by the sampling based tools, since sampling might

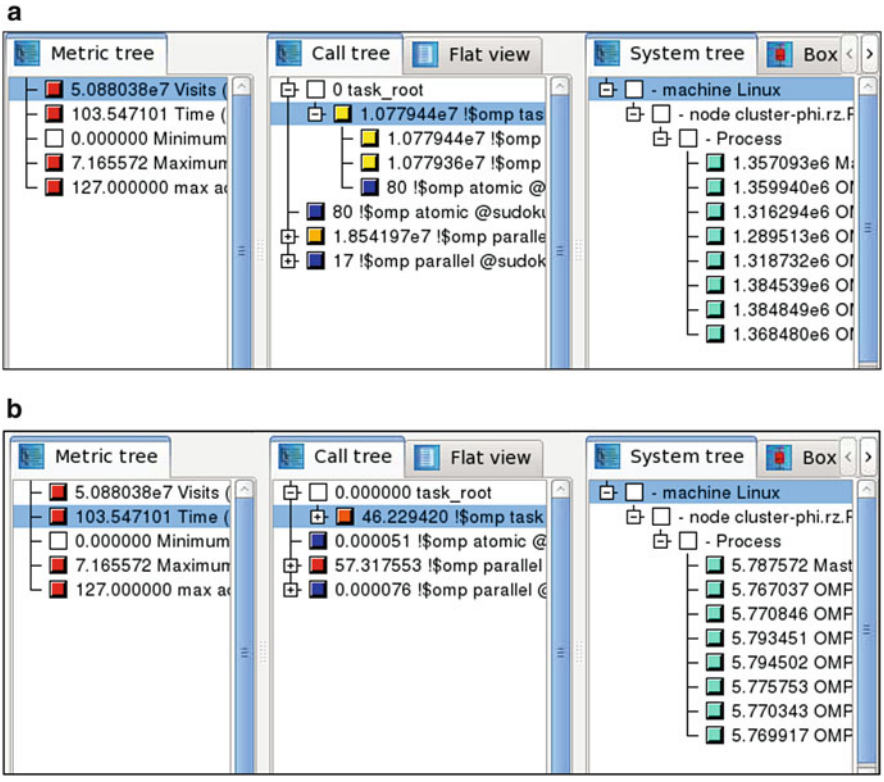


Fig. 3.5 Analysis results of the Sudoku solver with Score-P in profling mode. Visualization is done with the cube GUI. (**a**) Number of visits. (**b**) Time

miss task instances depending on the sampling rate. However, Score-P is not able to show information on a finer granularity as an instrumented region, so we cannot see source line information. Since Score-P can also not get information from the runtime, we can only see how much time was spent in a barrier or taskwait construct, but we cannot determine which amount is overhead and which is waiting time.

3.3.1.4 Score-P/Tracing

In tracing mode Score-P measures and stores the most detailed information of all tools. Beside all information of the profiling mode, additional information on individual regions is stored. Figure 3.6 shows the Vampir timeline view, where

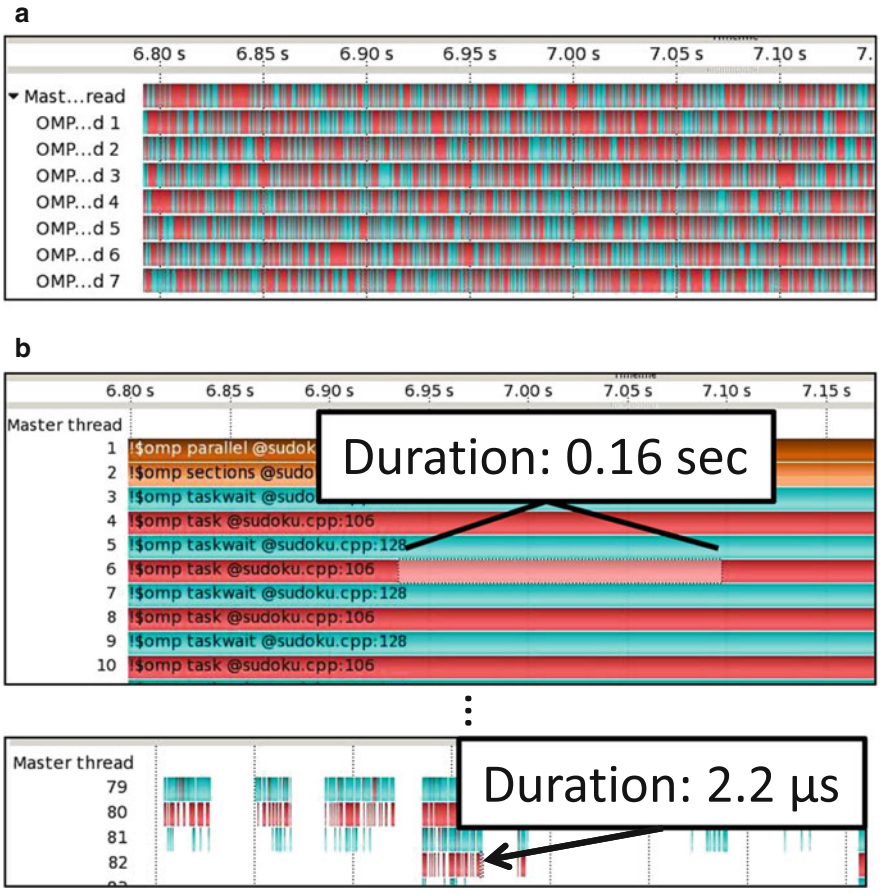


Fig. 3.6 Analysis results of the Sudoku solver with Score-P in tracing mode. Visualization is done with Vampir. (**a**) Timeline. (**b**) Callstack

we can see time spent in user code (red) and in taskwait regions (turquoise) on all threads. Looking more into details, the callstack view presents individual task-instances on different nesting levels. In Fig. 3.3b it can be observed, that a task in the second nested level has a duration of 0.16 s, whereas a deeper nested task has a duration of only 2.2 µs.

All tools show some overhead in the task, taskwait or barrier construct related to the task execution. The event based tools furthermore showed that the execution time of the tasks is very low and in the tracing tool we could also observe that tasks in the lower levels of the callstack are much smaller than higher ones. Therefore, we implemented a cut-off strategy for the Sudoku solver to stop creating tasks after the first two rows of the Sudoku board are processed. The resulting performance is also shown in Fig. 3.2. Obviously the performance problem is solved and the performance is much better in this optimized version, reaching a speedup of about 16 with 32 threads.

3.3.1.5 Overhead

The different measurement techniques (sampling or event based) and the different details of stored data (profile or trace) result in a different runtime overhead and amount of stored data. Table 3.2 shows the overhead and the amount of stored data for the different tools for a testrun with 16 threads. It can be seen that the overhead for the event based tools is really significant with about 150 %. The generated data should be interpreted with care, but as has been shown before the information provides useful hints on performance problems. Even if the average task duration would be 2 µs instead of the measured 4.6 µs, the problem would be more significant. The generated amount of data for the trace might also be a problem for larger programs, so our recommendation is to generate detailed traces for parts of the application which are critical. Other measurements then may help to identify the critical parts in advance.

Table 3.2 Overhead and generated amount of data for all investigated tools for the Sudoku code

16 Threads	Runtime overhead (setup routine)	Data volume (complete program)
Oracle Analyzer	<1 %	28 MB
Intel VTune	7 %	8.5 MB
Score-P (profiling)	∼150 %	44 KB
Score-P (tracing)	∼150 %	1.2 GB

3.3.2 Conjugate Gradient Method

Next we investigated a conjugate gradient solver (CG) where the sparse matrix vector multiplication was parallelized with OpenMP tasks. The tasks are spawned in a loop and one task processes `cs` rows of the matrix. The parameter `cs` can be adjusted to spawn more and smaller tasks, which is better for load balancing if the sparsity pattern is irregular, or to spawn only a few large tasks which introduces less overhead. We used all performance tools for different values of `cs`. Again all tools found the task region as a hotspot. The sampling tools also showed information on the source code level, but since the task region is only four lines long, this did not give any additional information. Again the event based tools delivered also the number of tasks and execution time for individual instances, but also this information was not very useful, since the number of matrix rows and the parameter `cs` is known, this can easily be computed by the programmer. For extremely large tasks, all tools identified load imbalance in the CG kernel. Overall, finding the best value for `cs` comes down to increase `cs` until the overhead spent in the OpenMP runtime is negligible. Since the sampling tools deliver this information directly due to the better cooperation with the vendor OpenMP runtimes, the sampling tools were preferable.

3.3.3 KegelSpan

The last investigated application is KegelSpan, a code simulating gearwheel cutting processes. KegelSpan is developed at the Laboratory for Machine Tools and Production Engineering at RWTH Aachen University. We investigated an experimental version of the code which has been parallelized with OpenMP tasks [6]. This experimental version differs from the production version. In the analyzed version a BSP tree is used to better handle the geometry which is adaptively refined at the cutting edge during the simulation. In particular the routine to setup the BSP tree recursively was investigated here. The problem of too small tasks has been observed for this code before and a cut-off mechanism was implemented which stops the creation of tasks at a certain configurable depth. We also investigated this application with all mentioned performance tools and found two performance weaknesses. First, with Score-P (tracing) we observed that even on the upper levels empty tasks are created. The reason therefore is that the BSP tree distributes the cell in two equal parts on every level. In some areas where only a few points reside, this can quickly lead to empty volumes, whereas at the refined area around the cutting edge a lot of points fall into an equal sized volume. We therefore changed the cut-off mechanism to stop creating tasks if the number of tasks in a cell is smaller than a certain threshold (version opt1). Second, the sampling based tools showed the hotspot on a sourcecode level at lines 4,717–4,744. In these lines points are resorted in an array. This process was never regarded as a hotspot and a simple

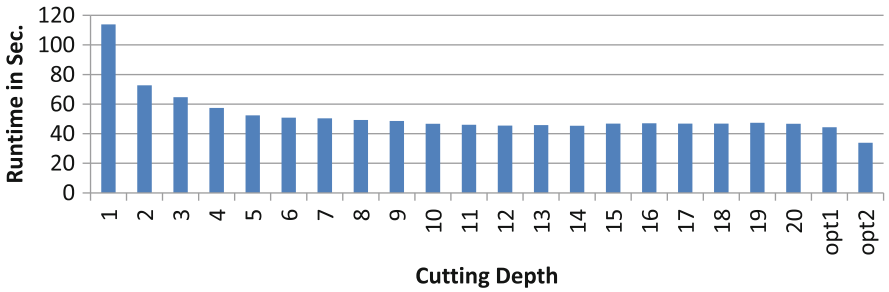


Fig. 3.7 Runtime of KegelSpan for different cut-off depths and both optimized versions opt1 and opt2

sorting strategy was used. We optimized this sorting to further optimize the routine (version opt2). The event based tools highlighted the task region as a hotspot which is several hundred lines in length, so the more detailed sampling based analysis was useful to find the hotspot here. Figure 3.7 shows the runtime of the original code version for different cut-off depths and both optimized versions. The optimized versions could save about 5 % (opt1) and additional 10 % (opt2) of execution time. The overhead for all investigated tools for this application was below 4 %, so even for the event based tools the observed runtime was nearly undisturbed.

3.4 Conclusion

In this work we looked at several codes employing OpenMP tasks to express parallelism and tried out different performance analysis tools to investigate the code performance and search for possible performance problems. The codes use recursive or iterative creation of tasks. Both sampling based tools presented roughly the same kind of information and the same level of detail. The strength of these tools was to give detailed statistical information at fine granularity, even on the source-code level. This information was proven useful to identify a hotspot within a task for the KegelSpan code. The information on the amount of runtime overhead was shown in both tools which helped to identify inefficient task constructs. The downside was that the construct was investigated, not the task-instance. When tasks created within the same construct had different behavior, which particularly happens with recursive task creation, this phenomenon was not observed by the tools. The event based tools provided more detailed information on a task-instance level which helped to identify the average task duration for profiling or even the duration of task-instances on different call-stack levels for tracing. This information also provided valuable insights in the execution behavior of the KegelSpan application. The overhead for event based tools was sometimes significantly higher than for sampling based tools, but still the information was very useful in many cases.

Acknowledgements Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IH11006 (LMAC).

References

1. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Parallel Processing, 2009 (ICPP '09), Vienna, pp. 124–131 (Sept 2009)
2. Fürlinger, K., Skinner, D.: Performance profiling for OpenMP tasks. In: Müller, M.S., Supinski, B.R., Chapman, B.M. (eds.) Evolving OpenMP in an Age of Extreme Parallelism. Lecture Notes in Computer Science, vol. 5568, pp. 132–139. Springer, Berlin/Heidelberg (2009).http://dx.doi.org/10.1007/978-3-642-02303-3_11
3. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture. In: Proceedings of the International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, pp. 51–65 (June 2008)
4. Gerndt, M., Ott, M.: Automatic performance analysis with periscope. Concurr. Comput.: Pract. Exp. **22**(6), 736–748 (2010)
5. Intel: Intel VTune Amplifier XE (Sept 2013). http://software.intel.com/en-us/intel-vtune-amplifier-xe
6. Kapinos, P., an Mey, D.: Productivity and performance portability of the OpenMP 3.0 tasking concept when applied to an engineering code written in Fortran 95. Int. J. Parallel Program. **38**(5–6), 379–395 (2010). http://dx.doi.org/10.1007/s10766-010-0138-1
7. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P – a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proceedings of 5th Parallel Tools Workshop, Dresden, (Sept 2011)
8. Lin, Y., Mazurov, O.: Providing observability for OpenMP 3.0 applications. In: Müller, M.S., Supinski, B.R., Chapman, B.M. (eds.) Evolving OpenMP in an Age of Extreme Parallelism. Lecture Notes in Computer Science, vol. 5568, pp. 104–117. Springer, Berlin/Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-02303-3_9
9. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. J. Supercomput. **23**(1), 105–128 (2002)
10. Nagel, W., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: visualization and analysis of MPI resources. Supercomputer **12**(1), 69–80 (1996)
11. Oracle: Oracle Solaris Studio 12.2: Performance Analyzer (Sept 2013). http://docs.oracle.com/cd/E18659_01/html/821-1379/
12. Shende, S., Malony, A.D.: The TAU parallel performance system, SAGE publications. Int. J. High Perform. Comput. Appl. **20**(2), 287–331 (2006)
13. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP tasking implementations on NUMA architectures. In: Chapman, B.M., Massaioli, F., Mller, M.S., Rorro, M. (eds.) OpenMP in a Heterogeneous World. Lecture Notes in Computer Science, vol. 7312, pp. 182–195. Springer, Berlin/Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-30961-8_14

Chapter 4
Recent Advances in Periscope for Performance Analysis and Tuning

Yury Oleynik, Robert Mijaković, Isaías A. Comprés Ureña, Michael Firbach, and Michael Gerndt

Abstract State of the art High Performance Computing (HPC) systems pose considerable programming challenges to application developers when tuning their applications. Periscope toolkit is one of a number of performance engineering instruments supporting application programmers in meeting those challenges. Due to the variety of architectures, programming models, runtime environments, and compilers on those systems, programmers need to apply multiple tools to understand and improve program performance. In this paper, we present the latest developments in Periscope aiming at (1) improving its interoperability and integration with other tools, (2) integrating automatic tuning support with performance analysis and (3) further extending performance analysis capabilities. The add-on for Periscope, called PAThWay, allows for the integration of multiple tools into performance tuning workflows. Further, Periscope is currently being extended with the ability to automatically tune parallel applications with respect to execution performance and energy consumption. And finally, new analysis capabilities were added to Periscope for the automatic evaluation of the temporal performance behavior of long-running applications.

4.1 Introduction

Programmers have always been required to write correct algorithms and, depending on the problem, this may already be challenging and time consuming. When working in projects where performance is part of the requirements, additional challenges rise from heterogeneity, non-uniform memory access and parallelism of the modern High Performance Computing (HPC) systems. Programmers need to find ways to map the software to the available memory and computing elements. Initially, the application is mapped to the hardware based on previous experience and guesswork.

Y. Oleynik (✉) • R. Mijaković • I.A. Comprés Ureña • M. Firbach • M. Gerndt
Institute of Informatics, Technical University of Munich (TUM),
Boltzmannstr. 3, 85748 Garching, Germany
e-mail: oleynik@in.tum.de

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_4

The tuning process of a parallel application involves several tools that specialize on different aspects that affect performance. These tools may be hardware or programming model specific. Even under a specific hardware and programming model, it is common that a tool will target only a subset of aspects that affect performance. For these reasons, programmers rely on a larger collection of tools to optimize their parallel applications.

The overall performance of a parallel application depends on its single node performance, its scalability and how well its load is balanced. On a single node, its performance will depend on the compilers available, the compiler flags and the programming models. In some systems, especially if heterogeneous, several compilers and programming models are used; this multiplies the tuning effort required (although it is not easy to quantify this effort, since it is a human factor). Typically, the scalability of the application depends on how much information has to be shared across nodes and the performance of the communication library and network. Finding a good partitioning scheme to balance the load and minimize communication is also essential.

Furthermore, the majority of scientific codes perform simulations iteratively, where a main loop of an application, sometimes referred to as the progress loop, executes the same computational kernel many times. Though the code being executed is often identical, application performance across the iterations vary significantly. As a consequence, the impact and location of performance bottlenecks, or degradations in this case, are time dependent. Taken into account that some simulations are run for several days and even month, performance analysis tools should be able to efficiently handle temporal dimension of performance measurements.

Recent developments in Periscope are focused on assisting application developers in overcoming the challenges described above. This is achieved

by enabling Periscope to answer the following typical user questions:

1. How to integrate the use of various tools in a single, well-defined workflow?
2. What is the optimal combination of tunable parameters for an application?
3. What are the most relevant runtime performance degradations?

In Sect. 4.2 we present PAThWay, which is part of the Periscope toolkit. It automates performance engineering workflows and is designed to methodically provide answers to the first question. Section 4.3 is dedicated to the Periscope Tuning Framework (PTF), which implements multiple specialized tuning plugins in order to automatically search for optima and thus answer the second question. Periscope's performance dynamics analysis, described in Sect. 4.4, answers the third question by searching for temporal degradations of performance in long running applications. We present an overview of the related work in Sect. 4.5 and, finally, we draw conclusions and perspectives for future work in Sect. 4.6.

4.2 Cross-Experiment Analysis and Tuning Using Workflows

A considerable challenge in optimizing HPC applications is the management of the overall optimization process. Which tools are used, and when? How is the collected performance data analyzed and archived? What were the performance characteristics of the application 3 weeks ago? With PAThWay, we are reaching out for the next level of integration, abstracting details of the HPC system and integrating the tools usage with human tasks.

PAThWay is a high-level tool that uses formal workflow definitions to provide structure to the overall optimization process. Major goals of this development are:

- Workflow automation
- System and Tools abstraction
- Experiment history

4.2.1 Workflow Automation

Whether formally defined or not, the process of performance tuning usually follows a specific workflow, which itself can be broken down into several activities. For example, typical activities involve making a snapshot of the source code for future reference, code instrumentation, job submission on the HPC system, etc. A simplified example of a performance tuning workflow is illustrated in Fig. 4.1.

The goal of the PAThWay development is to automate these activities to a large extent. This helps the application developer in two ways: First, by automating these repetitive and tedious tasks, time is saved and the number of unnecessary errors

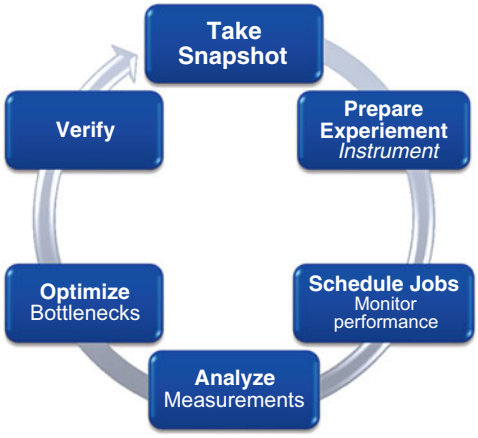


Fig. 4.1 Performance tuning workflow

is reduced. Second, the use of such automated workflows provides structure to the optimization process, encouraging methodical performance engineering.

Workflows can be defined using the standard Business Process Model and Notation (BPMN) format, for which there are graphical editors. Workflows have successfully been defined for common tasks, such as scalability analysis, cross-platform analysis and single-core profiling. Ideally, all activities that are performed during optimization, have a representation within the workflow. Once defined, the workflows can be *executed*, which triggers its individual activities. Workflow execution itself is performed by the jBPM workflow engine,[1] whereas the individual activities can invoke arbitrary tools or even human tasks.

By integrating all optimization steps in a single workflow, we emphasize structured and methodical performance engineering, which we believe is currently under-utilized in real-world HPC application development.

4.2.2 System and Tools Abstraction

The variety of tools that need to be integrated in an optimization workflow is a major complication for HPC application developers. The development of the joint measurement infrastructure Score-P [11] was motivated by this issue. Invoking tools from a workflow defined in PAThWay allows non-experts to use these tools and optimize their application, which more likely represents their actual area of expertise, e.g. physics simulation. In addition, workflows can use the tools to build higher-level constructs like scalability analysis, without requiring the user to have in-depth knowledge of the tools used on a low-level.

There are also several differences between HPC systems, which make cross-platform analysis difficult. We use the Parallel Tools Platform from Eclipse to provide the means to communicate with a variety of platforms. Jobs can be scheduled and monitored transparently on SLURM, LoadLeveler and MPICH2-enabled systems at the time of this writing.

Some tasks, such as analyzing tools output and optimizing the application, can only be done by humans. We call these tasks *human tasks* and alert the user when such a task needs to be performed according to the current state of workflow execution.

4.2.3 Experiment History

PAThWay records so-called *experiments* that result from workflow execution. An experiment is the notion of a job scheduled on an HPC system, its status and output,

[1] http://www.jboss.org/jbpm/

combined with a source snapshot and tools configuration. The experiment browser allows for later inspection of the experiments' outcome. There are plans for creating a variety of tools that can automatically draw conclusions from the experiment history in the future, such as analyzing cross-experiment performance dynamics.

4.3 Periscope Tuning Framework

Parallel software usually has several parameters that can be adjusted for better performance while keeping results correct. Examples of such parameters are compiler flags used during compilation and configuration parameters of communication libraries. Parallel software can also include tuning parameters, such as: algorithm selection, block sizes, partition factors, etc. There are also hardware settings that can be manipulated, such as performance governors and CPU frequencies (usually only available to administrators, but exposed in some systems through user level services).

The right combination of software and hardware parameters, that are best or at least good enough, is not always clear. There is typically a time consuming process of guesswork and collection of empirical data, that is done manually by the application developers. Given the large number of degrees of freedom, and therefore the large amount of time to empirically search for optimal or good combinations of these software and hardware parameter combinations, automated tools for this task are currently a necessity.

In this section we present an extension to Periscope, called the Periscope Tuning Framework (PTF). This framework allows for the implementation of plugins that automate the analysis and tuning process of parallel software. The plugins are built as shared objects that are loaded at runtime. With this approach, companies or individuals can develop custom plugins independently of the PTF core developers.

PTF follows the main Periscope principles, i.e., the use of formalized expert knowledge in form of performance properties and strategies, automatic execution, online search based on program phases, and distributed processing.

Periscope is extended by a number of tuning plugins that fall into two categories: online and semi-online plugins. An online tuning plugin is one that performs transformations to the application and/or the execution environment without requiring a restart of the application. In contrast, a semi-online tuning plugin will require one or more restarts of the application to achieve these effects.

Figure 4.2 illustrates the control flow of PTF. The tuning process is started with a preprocessing step. In this step, the application source files are instrumented and static analyses are performed on them. Periscope applies source level instrumentation for C/C++ and Fortran. The instrumenter generates a SIR file (Standard Intermediate Representation) that includes static information such as the instrumented code regions and their nesting information. The instrumentation and the static analysis process have been extended to support HMPP, OpenCL, and common parallel patterns (such as master-worker and pipelines).

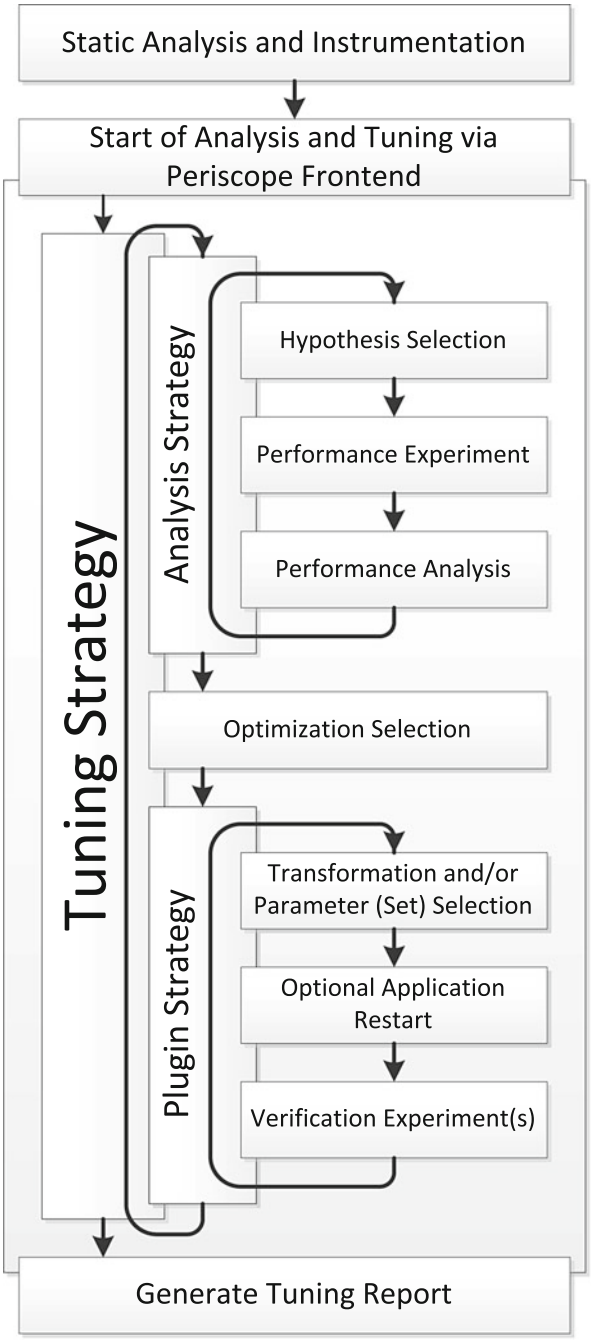


Fig. 4.2 Tuning control flow

The tuning is started via the Periscope frontend, either interactively or in a batch job. As done in Periscope, the application is started by the frontend along with the required agent hierarchy (given the size of the application and the hardware platform).

Periscope uses analysis strategies (e.g., MPI, OpenMP or single core analysis strategies) to guide the search for performance properties. This overall control strategy has now become part of a higher level tuning strategy. The tuning strategy controls the sequence of analyses and tuning steps.

Typically, an analysis can be used to determine application properties that will allow the plugin to restrict its search space. The search space can also be restricted by other plugin specific means, such as expert knowledge or machine learning approaches.

Once the search is done and an optimum or good enough solution is found, the tuning process is finished. A tuning report is then generated and presented to the user through standard output or files. This report documents the found performance properties as well as the tuning actions recommended by PTF. These tuning actions can be integrated into the application such that subsequent production runs will be more efficient.

Several plugins are being developed concurrently with the PTF. These include:

Compiler Flags Selection: Improves single node performance by evaluating possible valid compiler flags for a parallel application. The flags can be specified per file in the build and depend on the compilers available in the target system. For applications where their correctness depends on specific flags, these can be enforced.

High-Level Parallel Patterns for GPGPU Systems: Optimizes common parallel patterns on GPU based accelerators. Currently focuses on the pipeline pattern with OpenCL or CUDA. The total pipeline length and with of specific stages are optimized.

Hybrid Manycore HMPP Codelets: Evaluates the best selection of HMPP codelets for a specific application and hardware combination. In the HMPP application, several codelets that perform the same operation are prepared. The specific codelets and parameters are then evaluated and the best chosen.

Energy Consumption via DVFS: Performs a multi-objective optimization of performance and energy. For measurement, it relies on the use of hardware counters, timers and energy measurement hardware. The result selected will depend on the target desired performance and energy budget.

Master-Worker Pattern with MPI: Optimizes MPI applications that use the master-worker pattern. It evaluates the number of workers, the number of masters and the size of work packets.

MPI Runtime Parameters: Optimizes MPI communication performance for an application. It first analyses the application to find the location and MPI operations that take the most time. It then proceed to tune runtime parameters that are specific to these operations. The parameters include limits for the different communication protocols used by the library, as well as the selection of internal algorithms. Depends heavily on the available MPI implementations.

4.4 Runtime Performance Dynamics

Runtime Performance Dynamics Analysis is a new type of analysis supported by Periscope. It is based on dynamic phase profiling [13] and scale-space filtering [19] with consequent summarization of the resulting multi-scale representation. The algorithm can be split in the following steps: *measurements collection*, *preprocessing*, *qualitative summarization* and *property detection*. We explain the techniques used and the analysis algorithm in the following paragraphs.

4.4.1 Measurement Collection

Dynamic profiles collected using Score-P are used as an input for the algorithm. A dynamic profile is a time-series of measurements of some metric m sampled for each iteration of the application's progress loop, measured in a source-code region r on a process p. Such representations capture the evolution of the performance over the iterations of the simulation and attributes a time-series with a specific source-code location and process. An example dynamic profile time-series is plotted with blue color on Fig. 4.3.

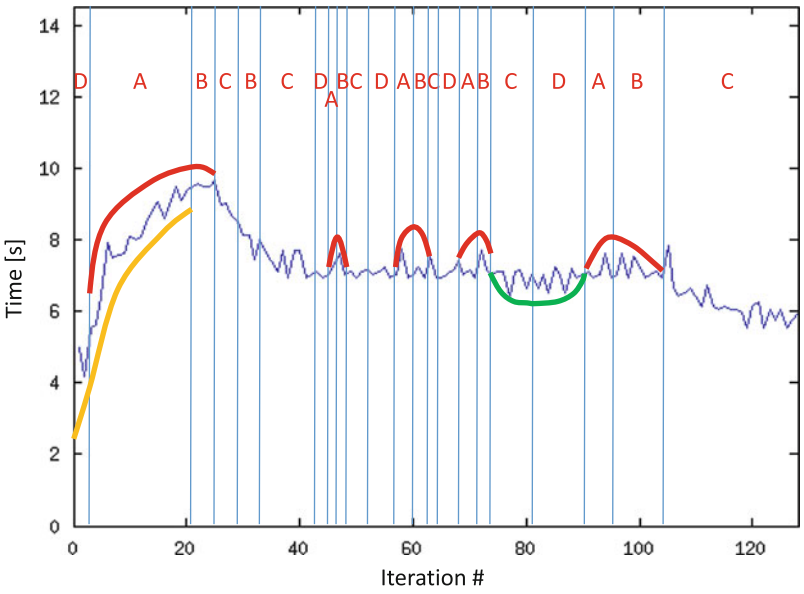


Fig. 4.3 Dynamic profile time-series overlaid with the extracted maximum-stability intervals and properties search results. The *red*, *green* and *yellow curves* highlight the results found for the queries "all peaks", "the most distinctive valley" and "the tallest increase", respectively

4.4.2 Preprocessing

Though the dynamic profiles capture dynamic behavior of the application performance, this information is implicit and requires manual interpretation by the user. A representation in terms of the intervals enclosed by zero-crossings of the first and second order derivatives offers an alternative qualitative semantic which is natural for the human comprehension. We apply alphabetical notation to denote the seven possible intervals defined by the combination of the signs of the first and second derivative: "concavely increase" – A, "concavely decrease" – B, "convexly decrease" – C, "convexly increase" – D, "linear increase" – E, "linear decrease" – F, "constant" – G.

However, the process of local extrema detection requires proper selection of the scale or the neighborhood over which the derivative is taken. A systematic way of handling the scale issue can be obtained by scale-space filtering. Scale-space filtering is a process of incremental low-pass filtering with the Gaussian filter of variable standard deviation, also known as the scale parameter. The resulting sequence of smoothed versions of the original time-series at different scales form a scale-space image. An important property of the scale-space image is that extrema persist over multiple scales and form a tree. Therefore, by detecting local extrema and connecting them from coarse to fine scales we can build a multi-scale hierarchy of intervals enclosed by two nearest extrema, called an interval tree.

4.4.3 Qualitative Summarization

In the next step we classify each interval in the tree to one of the seven possible shape types, defined above, and build a tree of qualitatively described segments. Each node of the tree is represented by a character denoting the shape of the interval and additional quantitative information, such as location, duration and magnitude of the corresponding interval.

It was shown empirically [19] that the number of scales over which an interval persists corresponds to the perceptual salience. We rely on this parameter, called stability, in order to quantify the relevance of the corresponding segments in the tree. Furthermore, we use the stability value to perform additional summarization by descending the tree in a breadth-first fashion and selecting the level with the maximum stability.

4.4.4 Automatic Search for Performance Dynamics Properties

The maximum stability level represents a dynamic profile time-series as a sequence of segments, each characterized by a qualitative shape descriptor, a stability value

Table 4.1 Example performance dynamics properties, corresponding queries and the color used to highlight resulting segment on the Fig. 4.3

Property	Query	Color
Local peaks	Select all "AB" sequences	Red
The most distinctive valley	Select "CD" sequence with max(stability)	Green
The tallest increase	Select "DA" sequence with max(magnitude)	Yellow

and supplementary quantitative information. Such representation allows efficient search for patterns specified with the following parameters:

- Qualitative shape specification (i.e. combination of shape descriptors)
- Perceptual salience (i.e. stability value)
- Magnitude and/or duration of the segment
- A combination of the parameters above

We demonstrate our technique on an example dynamic profile time-series plotted with blue color on the Fig. 4.3. The time-series represents samples of the metric "Execution time" measured for the body of the main loop for the first 128 iterations. The qualitative descriptors, represented with red characters, of the extracted maximum stability segments and their borders are overlaid on top of the time-series.

The proposed algorithm automatically answers the third question stated in the introduction section. In this case, the phrase "relevant degradation" can be defined in terms of the parameters described above. This allows very flexible specification of the performance dynamics properties, which can then be automatically searched in time-series. Table 4.1 shows three example properties and the corresponding queries. The resulting segments are highlighted with red, green and blue color on the Fig. 4.3.

4.5 Related Work

The complexity of today's parallel architectures has a significant impact on application performance. In order to avoid wasting energy and money due to low utilization of processors, developers have been investing significant time into tuning their codes. However, tuning implies searching for the best combination of code transformations and parameter settings of the execution environment, which can be fairly complicated. Thus, much research has been dedicated to the areas of performance analysis and auto-tuning.

The explored techniques, similar in approach to ours, can be grouped into the following categories:

- Tools that utilize techniques (such as optimization-space exploration [18], non-parametric inferential statistics [8], optimization orchestration [15] and machine

learning [6, 12]) to automatically analyze alternative compiler optimizations and search for their optimal combinations; and
- Auto-tuners (such as Active Harmony [4, 17] and MaSiF [5]) that search a space of application-level parameters that are known to impact performance. The search can be directed with the use of modeling or machine learning techniques [14]; and
- Frameworks that combine ideas from all the other groups (such as Autopilot [16] and the Insieme Compiler and Runtime infrastructure [10]).

Performance analysis and tuning are currently supported via separate tools. Periscope Tuning Framework aims at bridging this gap and integrating support for both steps in a single tuning framework.

Dynamic performance behavior is typically a subject for analysis using tracing techniques. However, due to scalability issues, profiling becomes a more attractive alternative. In particular, dynamic phase profile is used for performance dynamics evaluation and was first introduced in TAU [13]. Though the technique captures temporal performance evolution, the information about the relevant changes is implicit and requires manual interpretation by the user. In our work we build on top of this approach and apply scale-space filtering [19] with consequent qualitative summarization of dynamic profiles in order to automatically extract relevant performance changes. A different combination of signal processing algorithms is used in [3], where wavelet, morphological analysis and autocorrelation is used to extract periodical application phases from traces. Also in [7] clustering analysis and Multiple Sequence Alignment algorithm were used to automatically detect the computational structure of an SPMD program.

Using workflow engines to guide the overall performance tuning process and to integrate all individual tools with human tasks seems to be a rather unique approach within the HPC community. Although workflows are today more commonly used in scientific work in general [2], their use is much more common in business applications, which is why standards are usually centered around those. Popular standards include BPEL (Business Process Execution Language) [9], XPDL (XML Process Definition Language) and BPMN (Business Process Modeling and Notation) [1], of which the latter is also used by PAThWay.

4.6 Conclusion and Future Work

While working on performance analysis and optimization of HPC applications, programmers are faced with a wide spectrum of issues. These range from questions on how to organize performance experiments, choice of optimal configurations for the application and the runtime, to tracking and pinpointing dynamically appearing and disappearing performance bottlenecks. Recent developments in Periscope toolkit are aimed at assisting the programmer in handling these challenges by a number of technologies.

A new performance engineering automation tool, called PAThWay, is designed to assist the programmer in defining and automating the typical workflows involved in this process. The tool automates tasks like job submission, monitoring and retrieving results of the jobs on remote HPC systems; book-keeping of the carried out experiments and their configurations, maintenance of the documentation. In the future, we plan to extend the functionality of PAThWay for supporting analysis of cross-experiment performance dynamics.

In order to support programmers in configuring the application and the execution environment, Periscope is extended with a tuning interface and a number of plugins. The latter will automatically analyze application performance and based on the detected inefficiencies perform a search for the optimal parameters settings, where the parameters range from compiler flags to MPI runtime settings. In the future work, we plan to implement meta-plugins that will orchestrate other plugins in tuning multiple orthogonal parameter sets.

Finally, we present a new analysis algorithm which is aimed at supporting programmers in analyzing performance of long-running applications. A particular challenge of such analyses is that the location and severity of performance bottlenecks is time dependent. Using dynamic profiling and multi-scale analysis techniques we are able to search for complex patterns in the temporal performance behavior. For the future work we plan to extend the algorithm to search for similarities in temporal behavior observed on different processes of the parallel applications.

Acknowledgements The authors thank the European Union for supporting AutoTune project under the Seventh Framework Programme, grant no. 288038 and German Federal Ministry of Research and Education (BMBF) for supporting LMAC project under the Grant No. 01IH11006F.

References

1. Allweyer, T.: BPMN 2.0: Introduction to the Standard for Business Process Modeling. BoD–Books on Demand, Norderstedt (2010)
2. Barker, A., Van Hemert, J.: Scientific workflow: a survey and research directions. In: Parallel Processing and Applied Mathematics, pp. 746–753. Springer, Berlin/New York (2008)
3. Casas, M., Badia, R.M., Labarta, J.: Automatic phase detection and structure extraction of MPI applications. Int. J. High Perform. Comput. Appl. **24**(3), 335–360 (Aug 2010). http://dx.doi.org/10.1177/1094342009360039
4. Chung, I.H., Hollingsworth, J.: Using information from prior runs to improve automated tuning systems. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC '04, Pittsburgh, pp. 30–. IEEE Computer Society, Washington, DC (2004). http://dx.doi.org/10.1109/SC.2004.65
5. Collins, A., Fensch, C., Leather, H.: MaSiF: machine learning guided auto-tuning of parallel skeletons. In: Yew, P.C., Cho, S., DeRose, L., Lilja, D. (eds.) PACT, Minneapolis, pp. 437–438. ACM (2012). http://dblp.uni-trier.de/db/conf/IEEEpact/pact2012.html#CollinsFL12
6. Fursin, G., Kashnikov, Y., Wahid, A., Chamski, M.Z., Temam, O., Namolaru, M., Yom-tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C.: Milepost GCC: machine learning enabled self-tuning compiler (2009)

7. Gonzalez, J., Gimenez, J., Labarta, J.: Automatic evaluation of the computation structure of parallel applications. In: 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, Higashi Hiroshima, pp. 138–145. IEEE (2009)
8. Haneda, M., Knijnenburg, P., Wijshoff, H.: Automatic selection of compiler options using non-parametric inferential statistics. In: International Conference on Parallel Architectures and Compilation Techniques, Saint Louis, pp. 123–132 (2005)
9. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al.: Web Services Business Process Execution Language Version 2.0. OASIS Standard 11 (2007)
10. Jordan, H., Thoman, P., Durillo, J., Pellegrini, S., Gschwandtner, P., Fahringer, T., Moritsch, H.: A multi-objective auto-tuning framework for parallel codes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, Salt Lake City. IEEE Computer Society Press, Los Alamitos, pp. 10:1–10:12 (2012). http://dl.acm.org/citation.cfm?id=2388996.2389010
11. Knüpfer, A., Rössel, C., Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al.: Score-P: a joint performance measurement run-time infrastructure for periscope, scalasca, TAU, and vampir. In: Tools for High Performance Computing 2011, pp. 79–91. Springer, Berlin/Heidelberg (2012)
12. Leather, H., Bonilla, E.: Automatic feature generation for machine learning based optimizing compilation. In: Code Generation and Optimization (CGO), Seattle, pp. 81–91 (2009)
13. Malony, A.D., Shende, S.S., Morris, A.: Phase-based parallel performance profiling. In: G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E.Z. (eds.) Proceedings of the International Conference ParCo 2005, Malaga. NIC Series, vol. 33, pp. 203–210. John von Neumann Institute for Computing, Julich, (2006)
14. Nelson, Y., Bansal, B., Hall, M., Nakano, A., Lerman, K.: Model-guided performance tuning of parameter values: a case study with molecular dynamics visualization. In: International Parallel and Distributed Processing Symposium, Miami, pp. 1–8 (2008)
15. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), New York, pp. 319–332 (2006)
16. Ribler, R., Vetter, J., Simitci, H., Reed, D.: Autopilot: adaptive control of distributed applications. In: Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, pp. 172–179 (1998)
17. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.: A scalable auto-tuning framework for compiler optimization. In: International Parallel and Distributed Processing Symposium, Rome, pp. 1–12 (2009)
18. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.: Compiler optimization-space exploration. In: Proceedings of the international symposium on Code generation and optimization, San Francisco, pp. 204–215. IEEE Computer Society (2003)
19. Witkin, A.: Scale-space filtering: a new approach to multi-scale description. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP'84, San Diego, vol. 9, pp. 150–153. IEEE (1984)

Chapter 5
MuMMI: Multiple Metrics Modeling Infrastructure

Xingfu Wu, Valerie Taylor, Charles Lively, Hung-Ching Chang, Bo Li, Kirk Cameron, Dan Terpstra, and Shirley Moore

Abstract MuMMI (Multiple Metrics Modeling Infrastructure) is an infrastructure that facilitates systematic measurement, modeling, and prediction of performance and power consumption, and performance-power tradeoffs and optimization for parallel systems. MuMMI builds upon three existing frameworks: Prophesy for performance modeling and prediction of parallel applications, PAPI for hardware performance counter monitoring, and PowerPack for power measurement and profiling. In this paper, we present the MuMMI framework, which consists of an Instrumentor, Databases and Analyzer. The MuMMI Instrumentor provides automatic performance and power data collection and storage with low overhead. The MuMMI Databases store performance, power and energy consumption and hardware performance counters' data with different CPU frequency settings for modeling and comparison. The MuMMI Analyzer entails performance and power modeling and performance-power tradeoff and optimizations. For case studies, we apply MuMMI to a parallel earthquake simulation to illustrate building performance and power models of the application and optimizing its performance and power for energy efficiency.

X. Wu (✉) • V. Taylor • C. Lively
Department of Computer Science & Engineering, Texas A&M University, College Station, TX 77843, USA
e-mail: taylor@cse.tamu.edu

H.-C. Chang • B. Li • K. Cameron
Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA

D. Terpstra
Innovative Computing Lab, University of Tennessee, Knoxville, TN 37996, USA

S. Moore
Department of Computer Science, University of Texas at El Paso, El Paso, TX 79902, USA

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_5

5.1 Introduction

The burgeoning revolution in high-end computer architecture has far reaching implications for the software infrastructure of tools for performance measurement, modeling, and optimization, which has been indispensable to improved productivity in computational science over the past decade. Significant work has been done on exploring power reduction strategies for large-scale, parallel scientific applications [2–4,6,7,16]. The problem is that much of this work required manual data collection and analysis to explore the different power reduction techniques. Very little work has been done with respect to providing an infrastructure for automating as much of this process as possible in addition to archival of the data. The MuMMI (Multiple Metrics Modeling Infrastructure) [11] was developed to provide an infrastructure that facilitates systematic measurement, modeling, and prediction of performance and power consumption, and performance-power tradeoffs and optimization for multicore systems.

MuMMI is depicted in Fig. 5.1, as a multi-level infrastructure for integrated performance and power modeling for large-scale parallel systems. MuMMI builds upon an established infrastructure for performance modeling and prediction (Prophesy [20, 21, 25]), hardware performance counter measurement (PAPI [13]), and power profiling (PowerPack [4]). MuMMI extends these existing infrastructures in the following ways:

- Extension of Prophesy's well-established performance modeling interface to encompass multicores and to incorporate power parameters and metrics into the performance models and optimizations called E-AMOM (Energy-Aware Modeling and Optimization Methodology). A database component enables the modeling system to record and track relevant benchmark codes and results for characterizing multicore architectures.
- Building on top of PAPI's multi-component architecture, MuMMI provides a user-configurable layer for defining derived higher level metrics relevant to the performance modeling task at hand and mapping these to available native events on a given platform.

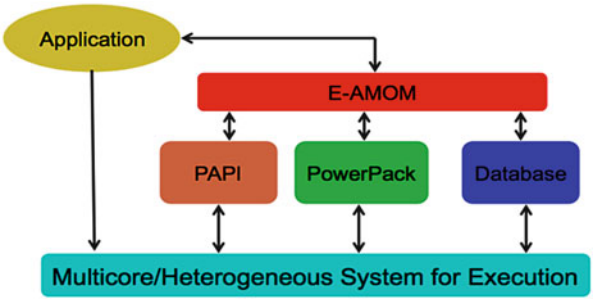


Fig. 5.1 Multiple metrics modeling infrastructure (MuMMI)

- Extension of an emerging power-performance measurement, profiling, analysis and optimization framework, called PowerPack, to multicore architectures. This work enables MuMMI to measure and predict power consumption at component (e.g. CPU, memory and disk) and function-level granularity for parallel architectures.

As a result of these combined efforts, MuMMI is able to offer an integrated and extensible performance modeling and prediction framework for use by application developers, system designers, and scientific application users, helping them to make key choices for configuring and selecting appropriate parallel systems and to evaluate and optimize power/performance on these systems.

In this paper, we discuss the MuMMI framework which consists of three main components: Instrumentor, Databases and Analyzer in detail. The MuMMI Instrumentor provides for automatic performance and power data collection and storage with low overhead. The Instrumentor has been used for our recent research work in [8–10]. MuMMI Databases extend the databases of Prophesy to store power and energy consumption and hardware performance counters' data with different CPU frequency settings. The MuMMI Analyzer (called E-AMOM) extends the data analysis component of Prophesy to support power consumption and hardware performance counters, and it entails performance and power modeling, and performance-power tradeoff and optimizations.

The remainder of this paper is organized as follows. Section 5.2 discusses the MuMMI framework in detail. Section 5.3 presents the MuMMI Instrumentor framework. Section 5.4 applies MuMMI to a parallel earthquake simulation to illustrate building performance and power models of the application and optimizing its performance and power for energy efficiency. Section 5.5 summarizes the paper.

5.2 MuMMI Framework

In this section, we discuss the four main components of the MuMMI framework shown in Fig. 5.1: PAPI, PowerPack, MuMMI database and E-AMOM.

5.2.1 PAPI

Hardware performance monitors can provide a window on the hardware by exposing detailed information on the behavior of functional units within the processor, as well as other components of the system such as memory controllers and network interfaces. The PAPI project designed and implemented a portable interface to the hardware performance counters available on most modern microprocessors [1, 13]. The PAPI specification includes a common set of PAPI standard events considered most relevant to application performance evaluation – cycle and operation counts,

cache and memory access events, cache coherence events, and branch prediction behavior – as well as a portable set of routines for accessing the counters. Moreover, performance monitors can now be found in memory controllers, network fabrics, and thermal monitors, as well as on special purpose accelerators such as GPUs. The work to restructure the PAPI library contains the exposed API and platform independent code, and a collection of independently loadable components, with one component for each set of hardware monitoring resources. As part of MuMMI, this work allows the simultaneous monitoring and correlation of performance data from multiple components and levels of the system.

5.2.2 PowerPack

PowerPack [4] is a collection of software components, including libraries and APIs, which enable system component-level power profiling correlated to application functions. PowerPack obtains measurements from power meters attached to the hardware of a system. The framework includes APIs and control daemons that use DVFS (dynamic voltage and frequency scaling) to enable energy reduction with little impact on the performance of the system. As multicore systems evolve, the framework can be used to indicate the application parameters and the system components that affect the power consumption on the multicore unit. PowerPack allows the user to obtain direct measurements of the major system components' power consumption, including the CPU, memory, hard disk, and motherboard. This fine-grained measurement allows power consumption to be measured on a per-component basis.

5.2.3 MuMMI Database

The MuMMI database facilitates contributions from and sharing among the high performance computing research community. In particular, MuMMI leverages from the Prophesy database [23, 24] that allows for web-based entry of data in addition to automatic upload of data via SOAP[17]. MuMMI database is the extension of the Prophesy database, which includes additional information relevant to power profiling, energy and hardware counters' performance on multicore platforms.

The MuMMI database schema is shown in Fig. 5.2. The MuMMI database is organized into four areas: application information, executable information, run information, and performance statistics. The application information includes details about the application and its developer. The executable information includes all of the entities related to generating an executable of an application. The run information includes all of the entities associated with running an executable and the system used for execution. Lastly, the performance statistics includes the performance data generated during the application execution.

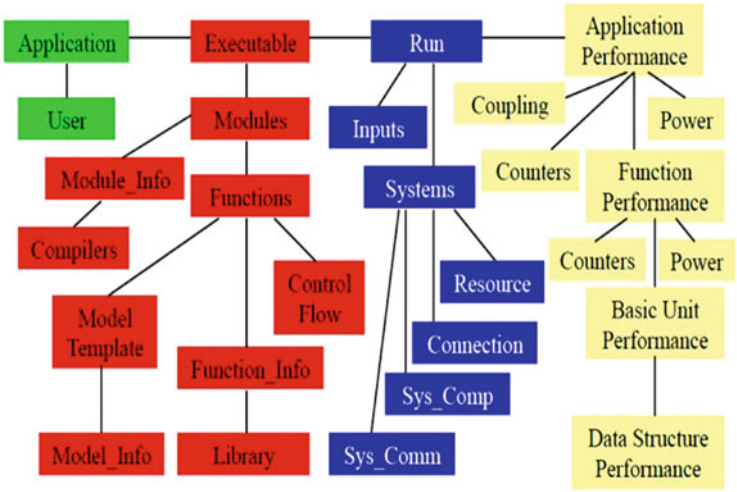


Fig. 5.2 MuMMI database schema

PAPI Multiplexing [13] allows a user to count more events than total physical counters simultaneously by a system. We collected the data for 40 hardware performance counters with different CPU frequency settings. The power and energy data from PowerPack includes system energy, CPU energy, memory energy, motherboard energy, disk energy, and power over time for each component (CPU, memory, motherboard and disk). The power and energy data are for an entire application or functions of the application.

5.2.4 E-AMOM

Based on hardware performance counters' data and power data from the MuMMI Database, we present the Energy-Aware Modeling and Optimization Methodology (E-AMOM) framework, which develops models of runtime and power consumption based upon performance counters and uses these models to identify energy-aware optimizations for scientific applications. These performance counters-based modeling methods [9,10] identify the different performance counters that are needed to accurately predict the application performance, and provides insight to improve the application performance.

During each application execution, we capture 40 performance counter events utilizing the PAPI. All performance counter events are normalized using the total cycles of the execution to create performance event rates for each counter. Using Spearman correlation and principal component analysis, we identify around five important performance counters for the multiple metrics of the application. The multiple metrics include runtime, system power consumption, CPU power consumption,

and memory power consumption. Then we use multivariable regression analysis to obtain the model equations for them based on these important performance counts and CPU frequencies.

E-AMOM utilizes these predictive models to employ run-time Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Concurrency Throttling (DCT) to reduce power consumption of the scientific applications, and uses cache optimizations to further reduce runtime and energy consumption of the applications.

5.3 MuMMI Instrumentor

Generally, collecting consistent performance and power data is difficult because it requires multiple executions of an application using each PAPI, PowerPack and Prophesy. Our goal for the MuMMI Instrumentor is to make performance and power data collection easy and automatic requiring only one application execution for the collection of performance and power data. To the best of our knowledge, this is the first tool to support automatic and simultaneous performance, power and energy data collection.

The MuMMI Instrumentor provides a way to do source-code level automatic instrumentation for Fortran77, Fortran90, C and C++ programs. It is an extension of Prophesy data collection component [23] with the addition of power and hardware counters data collection and dynamic CPU frequency scaling support. The basic framework of the MuMMI Instrumentor is illustrated in Fig. 5.3, which entails automatically inserting instrumentation code into a source code file. The Instrumentor supports the following instrumentation options:

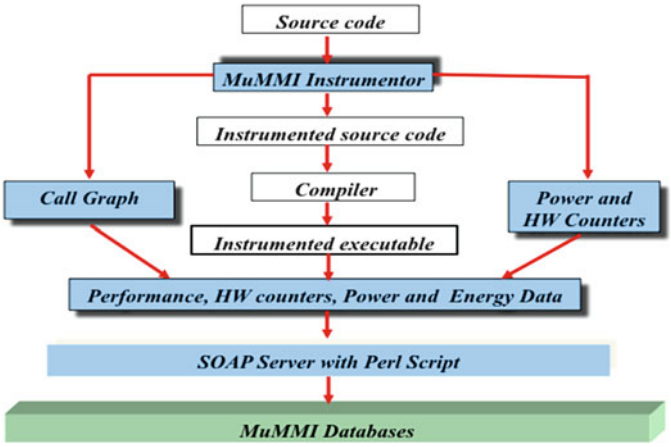


Fig. 5.3 MuMMI instrumentor framework

-a: Instrument all procedures and outer loops
-p: Instrument all procedures
-l: Instrument all loops
-n: Instrument all procedures not nested in loops
-F: Use Perl SOAP scripts to automatically transfer performance data to the MuMMI database
-W: Use PowerPack to collect power data
Default: Instrument procedures and outer loops

These instrumentation options (-a, -p, -l, -n, or default) can be used to instrument a source code at different levels based on the user, ranging from only procedures and outer loops to nested loops. The user also can select the source files to be instrumented. However, the following routines must be instrumented: the main file such as *main()* function for a C program or the *PROGRAM* for a Fortran program. The MuMMI Instrumentor counts each procedure and loop in the source code, and labels it with a unique instrumentation point number (or called event identifier).

For the case when the "-W" option is used, the MuMMI Instrumentor records the starting and end timestamps of the program, and at the end of a program execution, it sends these timestamps to PowerPack server to retrieve the power profiling data for the program, and appends the power and energy data to the output performance data files.

For the case when the "-F" option is used, at the end of the program execution, all performance data files are automatically transferred to the MuMMI database by the MuMMI Instrumentor. We use SOAP Lite package to develop Perl SOAP scripts to upload these data files to the MuMMI database (locally or remotely). Prior to invoking the SOAP client side to transfer the data files, the data files are converted to SQL scripts based on MuMMI database schema shown in Fig. 5.2 so that the SOAP server side receives the data files, and uploads the data to the MuMMI database. Further, the MuMMI Instrumentor is able to provide automatic upload of the information obtained from widely used performance analysis tools such as gprof, IPM [5], TAU [19] or Score-P [15] to the MuMMI database via some Perl SOAP scripts as well.

When the instrumented source code is generated, it requires the following compiler options to support different functionalities:

-DMPI: Support MPI programs
-DPAPI: Allow collecting hardware counters' data using PAPI
-DFS: Support dynamic frequency scaling

It also requires PAPI library link -lpapi to support collecting hardware counters' data using PAPI. If the "-DFS" option is used, the MuMMI Instrumentor can support dynamic frequency scaling by adjusting the CPU frequency to what a user sets in the system file scaling_setspeed. It requires the user to specify CPU frequencies for scaling up or down in advance.

5.4 Case Studies

In this section, we apply MuMMI to a parallel earthquake simulation to illustrate building performance and power models of the application and optimizing its performance and power for energy efficiency.

5.4.1 A MuMMI Testbed and a Parallel Earthquake Simulation

In this section, we describe a MuMMI testbed system, and deploy the MuMMI Instrumentor on the system. Our experiments utilize one multicore system from Virginia Tech, SystemG [18], which has 325 Mac Pro computer nodes. This system is the largest PowerPack-enabled research system with each node containing more than 30 thermal sensors and more than 30 power sensors. Each node has two quad-core 2.8 GHz Intel Xeon processors and 8 GB memory. The CPU frequency for the Intel Xeon processor can only be adjusted to 2.4 and 2.8 GHz.

Our experiments utilize a parallel earthquake simulation developed in [26, 27] with the SCEC/USGS benchmark problem TPV210. The benchmark problem TPV210 is the convergence test of the benchmark problem TPV10 [22]. In TPV10, a normal fault dipping at 60° (30 km long along strike and 15 km wide along dip) is embedded in a homogeneous half space. Pre-stresses are depth dependent and frictional properties are set to result in a sub-shear rupture. In the benchmark problem TPV210, we conduct the convergence test of the solution by simulating the same problem at a set of element sizes, i.e., 200, 100, 50 m, and so on, where m stands for meters. The simulation is memory-bound. Our hybrid MPI/OpenMP finite element earthquake simulations discussed in [26, 27] target the limitation to reduce large memory requirements. For the sake of simplicity, we only use TPV210 with 200 m element size in this paper.

5.4.2 Performance and Power Modeling

In this section, utilizing MuMMI Instrumentor, we develop application-centric models for the runtime and power consumption of the system, CPU, and memory.

To build a model, we create a typical training set for various configuration points first. The configuration M × N stands for M nodes with N cores per node. In the case of our hybrid MPI/OpenMP finite element earthquake application with the resolution of 200 m, M is the number of nodes and N is the number of cores with one OpenMP thread per core. For the hybrid application, a configuration 2 × 8 means that 2 MPI processes with 8 OpenMP threads per node with a total of 16 cores used.The training set for inter-node performance uses six data points consisting of configurations for points at 1 × 8, 3 × 8, 5 × 8, 7 × 8, and 9 × 8, 10 × 8. Performance

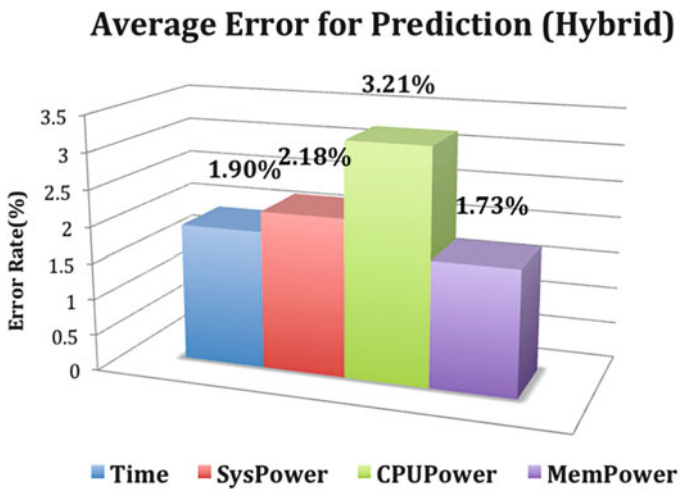


Fig. 5.4 Average prediction error rate using models for the hybrid earthquake simulation

counters' data and power profiling data for these training points are used to build performance and power models, then these models are used to predict performance and power consumption on larger number of processor cores (up to 64×8).

Figure 5.4 shows the average prediction error rate resulting from the modeling of the hybrid earthquake application. The average error rate for predicting the application runtime is 1.90 %.The average error rate for predicting the system power consumption is 2.18 %.The average error rate for predicting the CPU power consumption is 3.21 %.The average error rate for predicting the memory power consumption is 1.73 %. Overall, the models for the hybrid application have an average prediction error of less than 4 %.

Figure 5.5 shows the average prediction error rate resulting from the modeling of the MPI-only earthquake application. The performance components modeled for the MPI-only earthquake application had the smallest error for predicting the system power consumption (1.56 %) and the highest error rate occurred in modeling the memory power consumption (4.18 %).

Overall, our modeling method identifies the different performance counters that are needed to accurately predict the application performance and power consumptions.

5.4.3 Performance-Power Trade-off and Optimization

In this section, we incorporate two common software-based approaches for reducing power consumption in large-scale parallel systems, Dynamic Voltage and Frequency Scaling (DVFS)[6] and Dynamic Concurrency Throttling (DCT) [2]. DVFS can be

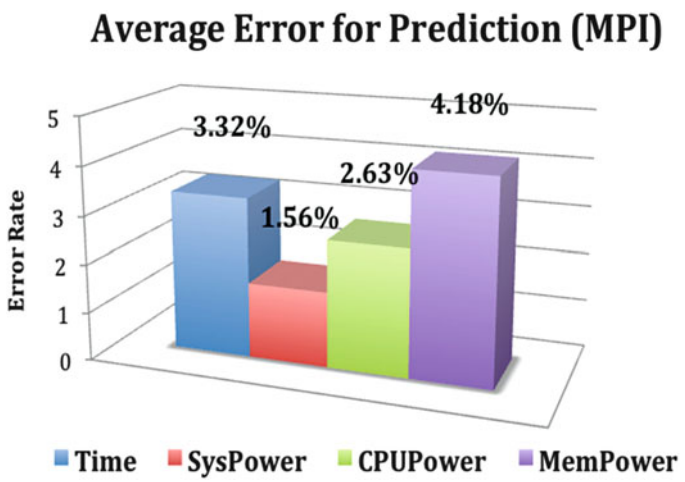


Fig. 5.5 Average prediction error rate using models for the MPI-only earthquake simulation

used to scale down the CPU frequency for a HPC application's workload resulting in lower power consumption. DVFS is most beneficial when applied to regions within an application where communication does not overlap with computation. DCT is used to control the number of threads assigned to a segment of a parallel code. DCT can be applied to a code region with a reduced workload that would not benefit from using the maximum number of cores on a chip.

Before applying DCT and DVFS to the application executed on SystemG, we investigated how the settings of having 1, 2, 4, 6, or 8 OpenMP threads per node impact the application performance by utilizing the MuMMI Instrumentor to collect the performance and power consumption data, and found using 2 OpenMP threads per node for the functions qdct3 and hourglass results in the best application performance. So DCT is applied to the functions hourglass and qdct3 so that they are executed using 2 OpenMP threads per node during the application execution. DVFS is applied to the functions input, communication, and final for the communication slacks by adjusting CPU frequency from 2.8 to 2.4 GHz.

As we discussed before, the hybrid earthquake application is memory bound. From our experiments, we found that L2 cache behavior is a major factor for memory power consumption. So additional loop optimizations such as using loop blocking with a block size of 8×8 and unrolling nested loops four times are applied to the application in order to improve cache utilization.

Figure 5.6 shows the performance, power, and energy consumption comparison. 2×8 stands for 2 nodes with 8 cores per node. For the hybrid application executions, we use 1 MPI process per node with 8 OpenMP threads per node as default. "Optimized-Hybrid" stands for applying DVFS, DCT and loop optimization to the hybrid application for execution. "System Energy" means the total energy consumed by the number of nodes used, and its unit is kilojoule. "System Power" means

#Cores	Program Type	Runtime(s)	System Energy (KJ)	System Power (W)	CPU Power (W)	Memory Power (W)
2x8	Hybrid	3156	1760.72	557.9	187.78	200.3
	Optimized-Hybrid	2980 (-5.9%)	1568.62 (-12.25%)	526.38 (-5.98%)	160.38 (-17.2%)	195.08 (-2.67%)
3x8	Hybrid	2166	1807.47	834.48	277.26	297.63
	Optimized-Hybrid	2031 (-6.65%)	1583.52 (-14.14%)	779.67 (-7.03%)	248.61 (-11.52%)	288.39 (-3.20%)
4x8	Hybrid	1681	1895.64	1127.68	375.56	404.12
	Optimized-Hybrid	1559 (-7.83%)	1639.4 (-15.63%)	1051.56 (-7.24%)	332.76 (-12.86%)	391.56 (-3.21%)
8x8	Hybrid	839	1900.32	2264.96	736.56	805.92
	Optimized-Hybrid	783 (-7.15%)	1656 (-14.75%)	2114.96 (-7.09%)	640.96 (-14.91%)	787.28 (-2.37%)
16x8	Hybrid	458	2117.76	4624.48	1506.72	1621.28
	Optimized-Hybrid	422 (-8.5%)	1789.28 (-18.35%)	4240 (-9.1%)	1379.04 (-9.25%)	1597.28 (-1.5%)
32x8	Hybrid	261	2411.84	9241.28	2900.16	3204.16
	Optimized-Hybrid	246 (-6.1%)	2055.36 (-17.34%)	8355.52 (-10.6%)	2694.08 (-7.65%)	3116.16 (-2.82%)
64x8	Hybrid	151	2693.12	17834.88	5703.04	6366.08
	Optimized-Hybrid	145 (-4.14%)	2318.72 (-16.15%)	15992.96 (-11.52%)	4753.28 (-19.98%)	6273.28 (-1.48%)

Fig. 5.6 Performance, power and energy consumption comparison

the total power consumed by the number of nodes used. "CPU Power" means the total power consumed by all CPUs used. "Memory Power" means the total power consumed by all memory components used. Overall, we reduce the execution time and lower power consumption of the hybrid application by applying DVFS, DCT and loop optimizations. This is a good improvement in performance and power consumption.

As shown in Fig. 5.6, the Memory Power consumption is larger than the CPU Power consumption for the hybrid earthquake application because the application is memory-bound, and applying DVFS, DCT and loop optimizations to the hybrid application benefit the CPU power consumption more than the Memory power consumption because we used 2 OpenMP threads per node when applying DCT to the hybrid application. It means that there are 6 idle cores per node during the execution of the dominated function qdct3 and hourglass. This results in a big CPU power saving. However, loop optimizations just improve cache utilization a little bit. For instance, on 512 cores (64×8), the optimized hybrid application execution results in 4.14 % improvement in Runtime, 16.15 % improvement in System Energy consumption, 11.52 % improvement in System Power consumption,

19.98 % improvement in CPU Power consumption, and 1.48 % improvement in Memory Power consumption. So the System Energy has the best improvement percentage because the System Energy is the product of Runtime and System Power. Overall, applying DVFS, DCT and loop optimizations to the hybrid MPI/OpenMP earthquake application reduced up to 8.5 % the execution time and lowered up to 18.35 % energy consumption.

5.5 Summary

In this paper, we discussed the MuMMI framework which consists of an Instrumentor, Databases and Analyzer. The MuMMI Instrumentor provides for automatic performance and power data collection and storage with low overhead. The MuMMI Databases store performance, power and energy consumption and hardware performance counters' data with different CPU frequency settings for modeling and comparison. The MuMMI Analyzer entails performance and power modeling and performance-power tradeoff and optimizations. For case studies, we applied MuMMI to a parallel earthquake simulation to illustrate building performance and power models of the application and optimizing its performance and power for energy efficiency. Further work will extend MuMMI Instrumentor to support additional power management techniques such as IBM EMON API on BlueGene/P/Q [28], Intel RAPL[14], and NVIDIA's Management Library for power measurement [12], and will focus on performance and power modeling for CPU + GPU systems and exploring the use of correlations among performance counters to provide further optimization insights.

Acknowledgements This work is supported by NSF grants CNS-0911023, CNS-0910899, and CNS-0910784.

References

1. 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**(2), 189–204 (2000)
2. Curtis-Maury, M., Dzierwa, J., et al.: Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: The International Conference on Supercomputing, Santa Fe (2006)
3. Ge, R., Feng, X., Cameron, K.W.: Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In: IEEE/ACM SC 2005, Seattle (2005)
4. Ge, R., Feng, X., Song, S., et al.: PowerPack: energy profiling and analysis of high-performance systems and applications. IEEE Trans. Parallel Distrib. Syst. **21**(5), 658–671 (2010)
5. Integrated Performance Monitoring (IPM). http://ipm-hpc.sourceforge.net/
6. Kappiah, N., Freeh, V., Lowenthal, D.: Just in time dynamic voltage scaling: exploiting inter-node slack to save energy in MPI programs. In: The 2005 ACM/IEEE Conference on Supercomputing (SC05), Seattle (2005)

7. Li, D., de Supinski, B., Schulz, M., Cameron, K., Nikolopoulos, D.: Hybrid MPI/OpenMP power-aware computing. In: Proceedings of the 24th IEEE International Conference on Parallel & Distributed Processing Symposium, Atlanta, May 2010
8. Lively, C., Wu, X., Taylor, V., Moore, S., Chang, H., Cameron, K.: Energy and performance characteristics of different parallel implementations of scientific applications on multicore systems. Int. J. High Perform. Comput. Appl. (IJHPCA) **25**(3), 342–350 (2011)
9. Lively, C., Wu, X., Taylor, V., Moore, S., Chang, H., Su, C., Cameron, K.: Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems. Comput. Sci. Res. Dev. **27**(4), 245–253 (2012). Springer
10. Lively, C., Taylor, V., Wu, X., Chang, H., Su, C., Cameron, K., Moore, S., Terpstra, D.: E-AMOM: an energy-aware modeling and optimization methodology for scientific applications on multicore systems. In: International Conference on Energy-Aware High Performance Computing, Dresden, Sep 2–3, 2013
11. Multiple Metrics Modeling Infrastructure (MuMMI) project. http://www.mummi.org
12. NVIDIA, NVIDIA's Management Library (NVML) API Reference Manual (2012)
13. PAPI (Performance API). http://icl.cs.utk.edu/papi/
14. Rotem, E., Naveh, A., Rajwan, D., Ananthakrishnan, A., Weissmann, E.: Power-management architecture of the intel microarchitecture code-named sandy bridge. IEEE Micro. **32**(2), 20–27 (2012)
15. Score-P, Scalable Performance Measurement Infrastructure for Parallel Codes. http://www.vi-hps.org/projects/score-p/
16. Singh, K., Bhadhauria, M., McKee, S.A.: Real time power estimation and thread scheduling via performance counters. In: Proceedings of the Workshop on Design, Architecture, and Simulation of Chip Multi-Processors, Lake Como, Nov 2008
17. SOAP (Simple Object Access Protocol). http://www.w3.org/TR/soap/
18. SystemG at Virginia Tech. http://www.cs.vt.edu/facilities/systemg
19. TAU (Tuning and Analysis Utilities). http://www.cs.uoregon.edu/research/tau
20. Taylor, V., Wu, X., Geisler, J., Stevens, R.: Using kernel couplings to predict parallel application performance. In: Proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2002), Edinburgh, 24–26 July 2002
21. Taylor, V., Wu, X., Stevens, R.: Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. ACM SIGMETRICS Perform. Eval. Rev. **30**(4), 13–18 (2003)
22. The SCEC/USGS Spontanous Rupture Code Verification Project. http://scecdata.usc.edu/cvws
23. Wu, X., Taylor, V., Stevens, R.: Design and implementation of prophesy automatic instrumentation and data entry system. In: The 13th International Conference on Parallel and Distributed Computing and Systems (PDCS2001), Anaheim (2001)
24. Wu, X., Taylor, V., et al.: Design and development of prophesy performance database for distributed scientific applications. In: Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, Virginia (2001)
25. Wu, X., Taylor, V., Geisler, J., Stevens, R.: Isocoupling: reusing coupling values to predict parallel application performance. In: 18th International Parallel and Distributed Processing Symposium (IPDPS2004), Santa Fe (2004)
26. Wu, X., Duan, B., Taylor, V.: Parallel simulations of dynamic earthquake rupture along geometrically complex faults on CMP systems. J. Algorithm Comput. Technol. **5**(2), 313–340 (2011)
27. Wu, X., Duan, B., Taylor, V.: Parallel earthquake simulations on large-scale multicore supercomputers, (Book Chapter). In: Furht, B., Escalante, A. (eds.) Handbook of Data Intensive Computing. Springer, New York (2011)
28. Yoshii, K., Iskra, K., Gupta, R., Beckman, P., Vishwanath, V., Yu, C., Coghlan, S.: Evaluating power monitoring capabilities on IBM Blue Gene/P and Blue Gene/Q. In: IEEE Conference on Cluster Computing, Beijing (2012)

Chapter 6
Cudagrind: Memory-Usage Checking for CUDA

Thomas M. Baumann and José Gracia

Abstract The Memcheck tool, build on top of the popular Valgrind framework, offers a reliable way to perform memory correctness checking in arbitrary x86 programs. At runtime Valgrind dynamically transforms the program into intermediate code which is then being executed on an emulated CPU. Due to a full shadow copy of the memory used by each program the Memcheck tool is able to perform various, bitwise precise runtime checks. These include, but are not limited to, detection of memory leaks, checking the validity of memory accesses and tracking the definedness of memory regions. But despite the wide applicability of this approach, it is bound to fail when accelerator based programming models are involved. Kernels running on such devices, like NVIDIA's Tesla series, are completely separated from the host. The memory on the device is only accessible through an API provided by the driver or from inside the kernels. Due to this indirect approach Valgrind is not able to understand, instruments or even recognize memory operations being executed on the device. Freeing Valgrind from this limitation has been the focus of the work presented here. A set of wrappers for a subset of the CUDA driver API has been introduced. These allow tracking of (de-)allocation of memory regions on the device as well as memory copy operations needed to place and retrieve data in device memory. This provides the ability to check whether memory is fully allocated during a transfer and, thanks to the host memory checking performed by Valgrind, whether the memory transferred to the device is fully defined and addressable on the host. This techniques allows detection of a number of common programming mistakes, many of which can be rather difficult to debug by other means. These wrappers, combined with Valgrind's Memcheck tool, is being called Cudagrind.

6.1 Introduction

Many parallel programming models require the user to provide so-called memory buffers which are used as transient storage area for message passing. Similar usage of buffers appears in programming accelerators, like GPGPUs, where data

T.M. Baumann (✉) • J. Gracia
High Performance Computing Center Stuttgart, Nobelstr. 19, 70565 Stuttgart, Germany
e-mail: baumann@hlrs.de

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_6

is transferred between host and accelerator device. Maintaining consistency of such memory operations in an asynchronous environment is a complex task. Failure to properly manage the usage of memory buffers can give rise to erratic and undefined behavior. This includes, but is not limited to, reading data from undefined memory regions and crashes from writing into forbidden parts of the memory. In a worst case scenario, incorrect usage of memory buffers can even lead to serious security concerns. In the right circumstances not only the user's applications but also the operating system might be affected when memory buffers are shared between different parts of the system.

One approach for debugging these kinds of memory operations are runtime correctness-checks, e.g. by using frameworks like Valgrind [4] and specifically it's tool Memcheck. A solid understanding of their correct usage can potentially save hundreds of hours spent on locating subtle problems like a misaligned pointer or prematurely freed memory regions. Despite the importance of runtime correctness-checking, however, the existing tools do not support accelerators as for instance CUDA-based GPGPUs. In such a setting, so called kernels are running on the accelerator device and operate on their own dedicated device memory, which is not shared with the host. The device memory can be accessed from the device's compute cores with a high bandwidth which, together with the manyfold parallelism of the GPGPU, gives rise to exceptional performance boost for certain computational problems. On the down side, however, CUDA programs are very difficult to debug, as kernels running on the device are not accessible to typical debugging tools, nor in fact the main program, that are running on the host. The host can access memory on the device only indirectly through a specific set of API[1] functions offered by the CUDA driver. Due to this indirect access pattern classical tools, like Valgrind, are unable to work with and on device memory. While there are some tools that work on the device, e.g. NVIDIA's cuda-memcheck, they often do not check the memory transactions between host and device.

In this paper, we present **Cudagrind**, a tool which relying on the rich features of Valgrind aims to fill the gap of correctness-checks on the host, the device, and transaction between those.

6.2 Memory Checking

Manually checking source code, even if only a couple of handful of lines against only a few most common memory buffer related programming errors, is already a complex and tedious task. Doing the same for hundreds of thousands of lines of code is next to impossible. There is a wide range of tools available that aim to assist the programmer in such a task. In general these fall into one of three categories: (1) static analysis tools, as for instance Lint [3], which work on the source level, (2) runtime

[1] http://docs.nvidia.com/cuda/cuda-driver-api/index.html

correctness-checkers, as for instance Valgrind [5], that examine buffer usage during execution, and (3) post-mortem analysis as offered by some tracing tools or online debugging after the fact. Naturally, these tools cannot offer full protection against memory-related errors, else they would solve the *halting problem*; however, they offer a simple practical method to increase the confidence in one's own code.

6.2.1 Valgrind: Heavy Weight Dynamic Binary Instrumentation

Valgrind is a framework for dynamic binary instrumentation of arbitrary programs. In a nutshell, it consists of a just-in-time compiler, which translates the program's binary into Valgrind's *intermediate representation (IR)* as the program executes. This architecture-neutral IR is then executed on an emulated generic x86 processor. Thus, the JIT compiler makes sure that every machine instruction in the original binary is translated into an equivalent sequence of IR instructions to be executed on the virtual processor. This approach, in principle, allows for dynamic runtime instrumentations of arbitrary architectures, but requires a functional JIT compiler for each target architecture.

Tools based on the Valgrind framework, as for instance the well known Memcheck tool [5], can then use the dynamic instrumentation for any kind of runtime checks. Specifically, Memcheck keeps a full shadow copy of the application's memory and registers. Access to memory regions and de-/allocation of memory is tracked in the shadow copy. This allows various checks on every memory or register access. For example, an access, to a memory region which has not yet been marked as allocated, will be reported as an error; similarly, reading from a memory region, which has been allocated but not yet initialized, will also be reported as an error. Having a full shadow copy allows Memcheck to operate with bitwise precision.

A serious disadvantage, though, is the large runtime overhead incurred by this approach. The emulated code is executed by the Valgrind runtime already roughly four times slower compared to the native execution. When using a tool like Memcheck the overhead can results in a 100-fold slowdown of the application. Another limit of this approach is the additional memory required by the shadow memory, which leaves only half of the total memory available to the application.

6.3 Cudagrind Internals

The CUDA stack depicted in Fig. 6.1 shows the various ways of accessing a CUDA enabled device. Ultimately every access to a device is being reduced to a CUDA driver call. This holds true for programs based on the low level CUDA driver API as well as the streamlined CUDA runtime API. So a call to `cudaMemcpy(..)`

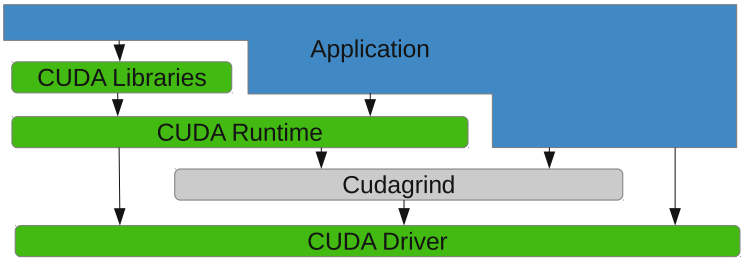


Fig. 6.1 The placement of Cudagrind in the CUDA stack. There is a wrapper for every memory related function on the CUDA driver level. Functions calls from an application on a higher level, e.g. a CUDA library function, are handled when the respective driver function is called from the library or runtime level

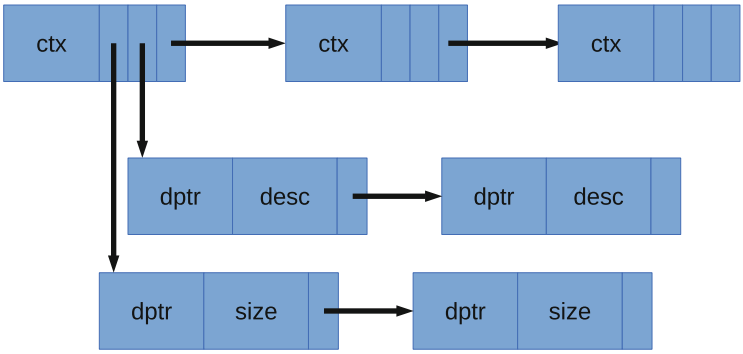


Fig. 6.2 Internal list of memory regions that have been allocated on the device. Each allocation generates a new entry into the list of linear memory or device arrays of the respective CUDA context ctx. Both lists contain information about the device pointer returned by the allocation and either the size of the linear memory or device array descriptor

implicitly leads to a call of any of the `cuMemcpyXtoY(..)` functions provided by the driver. Conveniently the same is true for programing models based on higher abstraction levels. Pragma based models, like for example OpenACC, rely on the functionality exposed by the CUDA driver or, implementation dependent, the CUDA runtime. Finally CUDA libraries rely on any of the aforementioned methods to access a device. Hence every access to a CUDA device will eventually reach the CUDA driver.

This makes the CUDA driver level a natural choice for the wrappers provided by Cudagrind. Every time a program handles device memory or transfers data between the device and/or the host the respective Cudagrind wrapper is being called instead. No matter the programing model or abstraction layer used. Depending on the requested operation several things will then happen. The internal list of memory regions allocated on the device being updated (Fig. 6.2). The parameters of the function called are cross checked against this list as well as Valgrind's knowledge about the host memory. This allows detection of various errors like

- Data being copied into or from unallocated memory regions.
- Undefined data being copied from host onto the device.[2]
- Copying more data than there's previously allocated memory.
- Concurrent access related to asynchronous operations.

6.4 Software Requirements

To compile the Cudagrind wrappers the following components need to be installed and available

- Valgrind 3.6.0 or newer.
- CUDA SDK 4.0 or newer.
- CUDA driver that works with installed SDK.
- GCC or compatible compiler that supports the GNU 'constructor' attribute.

Additionally the location of the respective driver's `libcuda.so` and Valgrind's `valgrind.h` needs to be known and passed to the linker when compiling Cudagrind.

To execute a program with the Cudagrind wrappers the same Valgrind version used during the compilation needs to be present. The generated `libcudagrind.so` and the `libcuda.so` from the respective CUDA driver needs to be added to `LD_PRELOAD` in this order. The addition of `libcuda.so` to `LD_PRELOAD` is only needed in cases where the original program has not been fully linked against it. This is the case when using certain OpenACC compilers, where running the program with the Cudagrind wrappers would lead to unknown references to functions called only from within the wrappers but not the main program itself.

Now the executable that is to be run with Cudagrind can simply be called with `valgrind <executable>`. Every time one of the wrapped functions in the CUDA driver is executed Valgrind will replace it with the respective Cudagrind wrapper and perform all needed checks at runtime. Optionally a suppression file can be generated and passed to Valgrind at runtime. This suppresses certain spurious errors reported by Valgrind in the CUDA driver.[3] In certain rare cases the lib path of the Valgrind installation `/path/to/valgrind/lib/valgrind` needs to be added to `LD_LIBRARY_PATH`.

[2] This might not be an error in cases where the host memory is accessed in a strided way.

[3] See the Valgrind manual that comes with your installation or at the Valgrind homepage at http://valgrind.org/ for more information about suppression files.

6.5 Usage Example

This section offers a brief overview about how Cudagrind is able to assist in detecting and solving memory related errors in a CUDA program. Let 'vecsum' in Listing 6.1 be a kernel that performs a basic vector addition of two input vectors `a` and `b` and writes the result into the vector `c`.

Listing 6.1 Basic vector sum in CUDA C

```
__global__ void vecsum(double *c, double *a, double *b) {
   start = threadIdx.x+(blockIdx.x*blockSize.x);
   size  = blockSize.x*grindSize.x;
   for (int i = start ; i < size ; i += blockSize.x) {
      c[i] = a[i]+b[i];
   }
}
```

Further assume the following excerpt from a program utilizing this kernel in Listing 6.1. Instead of enough memory to hold `size` values of type double, only an amount able to hold as many values of type float is being allocated.

Listing 6.2 Part of a CUDA C program utilizing the vecsum kernel from Listing 6.1. Note the faulty allocation of the result variable c using **float** instead of **double**

```
...
double *c, *a, *b;
double *c_host;
int size = 1000000;
...
cudaMalloc((void**)&c, size*sizeof(float));
cudaMalloc((void**)&a, size*sizeof(double));
cudaMalloc((void**)&b, size*sizeof(double));
...
vecsum<<<gridSize, blockSize>>>(c, a, b);
...
cudaMemcpy(c_host, c, size*sizeof(double),
           cudaMemcpyDeviceToHost);
...
```

The following sections show the results of executing this code with various types of error checking or lack thereof.

6.5.1 No Error Checking

Running the code without external tools or utilization of error values returned by the API might lead to this error going unnoticed. Formally the result of the kernel execution will be undefined due a buffer overflow (i.e. writing into the unallocated memory regions beyond the first `size/2` double values). Additionally the call to `cudaMemcpy` will fail, but not necessarily crash the program, due to the wrong amount of memory scheduled for copying. In the best case the undefined behavior will lead to a crash. But it is more likely for the bug to go unnoticed, leading to the

program's execution with the undefined values stored in the host memory pointed at by `c_host`. On the bright side, this method of error detection does not ask for recompilation of nor debug information being present in the binary and causes no overhead in execution time!

6.5.2 API Error Value Checking

The most basic, yet still often underutilized, way of error checking relies on the error values returned by any of the CUDA runtime API functions. This also works in the case of asynchronous operations, although special care has to be taken. Namely a call to either `cudaGetLastError` or `cudaPeekAtLastError` is needed after a synchronization point that guarantees the asynchronous execution being completed. To simplify the process in the synchronous case a preprocessor macro, like the one shown in Listing 6.3, can be used. For asynchronous operations a synchronization point would have to be added to the macro to get the error message generated by the operation.

Listing 6.3 Preprocessor macro to simplify runtime API error value checking. Use cudaVerify with every call to the CUDA runtime. Similar principle applies to CUDA driver API error checking. This and similar macros are included in the cutil.h header file of CUDA SDKs prior to 4.2

```
#define cudaVerify(x) do {                                  \
 cudaError_t __cu_result = x;                               \
 if (__cu_result!=cudaSuccess) {                            \
  fprintf(stderr,                                           \
     "%s:%i:␣error:␣cuda␣function␣call␣failed:\n" \
     "␣␣%s;\nmessage:␣%s\n",__FILE__,__LINE__,      \
     #x,cudaGetErrorString(__cu_result));                   \
  exit(1);                                                  \
 }                                                          \
} while(0)
```

With the help of `cudaVerify` the output shown in Listing 6.4 is being generated when the call to `CudaMemcpy` in Listing 6.2 happens. It contains the location in the source code where the error is being recorded. The runtime overhead is negligible in most cases, as the actual amount of manual calls to the CUDA runtime is small compared to the time spend in kernel executions, memory transfers and even operations on the host. On the other hand the 'invalid argument' error message, generated by `cudaGetErrorString`, is very undescriptive. This method is also rather error prone, as it relies on manual use of `cudaVerify` by the programmer with every call to any CUDA runtime function. Finally it needs to be adapted to be used with different CUDA based programing models, including the CUDA driver API.

Listing 6.4 Running the faulty code shown in Listing 6.2 with verbose runtime error checking produces the following output. Location of the error is shown, but the error message is non-descriptive

```
hpc43598 n093302 306$./a.out
example01.cu:60: error: cuda function call failed:
   cudaMemcpy(c_host, c,(vector_size)*sizeof(double),
cudaMemcpyDeviceToHost);
message: invalid argument
```

6.5.3 Valgrind/Memcheck

Utilizing only the Valgrind tool Memcheck produces no warnings or error messages, except possibly spurious errors inside the CUDA driver library `libcuda.so`. As mentioned in Sect. 6.4 these are not connected to actual errors in the program being run and can be either ignored or suppressed with a Valgrind suppression that's tailored to CUDA driver being used. In this case Valgrind is unable to detect the error since the problem only occurs on the device and inside the kernel being executed. This part of the program is effectively invisible to the Valgrind framework. At the current time it is unable to translate and run the PTX or CUBIN code that is being transferred to and run on the device by the CUDA driver.

6.5.4 Cuda-Memcheck

NVIDIA's tool `cuda-memcheck` provides an alternative for runtime memory checks. A given program is executed through `cuda-memcheck` and every error detected is printed to `stdout`. Conveniently this functionality is tightly integrated into `cuda-gdb`, NVIDIA's CUDA extension to the GNU debugger. Setting '`set cuda memcheck on`' inside a debugging session enables the same functionality provided by the standalone `cuda-memcheck`.

If being used on the code from Listing 6.2 `cuda-memcheck` produces the output shown in Listing 6.5. It reports the location of the error in the call stack as well as the internal error code reported by the CUDA runtime. Alas, the actual string representation or description of the error is missing from the output. As is the location in the actual source of the program, that's only reported as an address in the binary. The latter even holds true if the program has been compiled with `-g -G`, i.e. with full debug information.

Listing 6.5 Running the faulty code shown in Listing 6.2 with cuda-memcheck produces this output. Location of the error is only shown inside the binary and only the error number is reported

```
hpc43598 n093302 307$cuda-memcheck ./a.out
CUDA-MEMCHECK
Program hit error 11 on CUDA API call to cudaMemcpy
Saved host backtrace up to driver entry point at error
Host Frame:/../libcuda.so [0x26a180]
Host Frame:/../libcudart.so.5.0 (cudaMemcpy + 0x28c) [0x3305c]
Host Frame:./a.out [0xc55]
Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ecdd]
Host Frame:./a.out [0x9f9]
ERROR SUMMARY: 1 error
```

6.5.5 *Cudagrind*

Running the code snippet in Listing 6.2 through Cudagrind will produce the output shown in Listing 6.6. It contains a detailed error message that gives precise information about the encountered problem. The user learns that the target memory region on the device only features half the amount of memory needed for the operation. This gives a strong hint at the actual error in code, where the result vector `c` has been allocated for float instead of the needed double values. With added debug information the output also shows the exact location of the error in the source code as can be seen in Listing 6.6. Otherwise, as always the case when using `cuda-memcheck`, the call stack contains only reveals the address in the binary of the program.

Listing 6.6 Running the faulty code shown in Listing 6.2 with the Cudagrind wrappers produces this output. Location of the error in the source code is only shown when compiled with debug informations. A detailed error message gives a hint about the potential problem

```
hpc43598 n093302 307$valgrind ./a.out
Error: Allocated device memory too small for device->host copy.
Expected 8000000 allocated bytes but only found 4000000.
 at 0x4C18E79: VALGRIND_PRINTF_BACKTRACE (valgrind.h:4477)
 by 0x4C19261: cuMemcpyDtoH_v2 (cuMemcpyDtoH.c:58)
 by 0x5A1E531: ??? (in /../libcudart.so.5.0.35)
 by 0x5A40E43: cudaMemcpy (in /../libcudart.so.5.0.35)
 by 0x400C54: main (example01.cu:60)
```

6.6 Runtime Overhead

Using Valgrind and specifically Cudagrind always incurs a runtime overhead. While potential high in case of Valgrind, due to the JIT and emulation approach, Cudagrind itself has nearly no effect on the runtime of a program. Additionally for Valgrind the slowdown also depends on the ratio of pure CPU and GPU code being executed, as

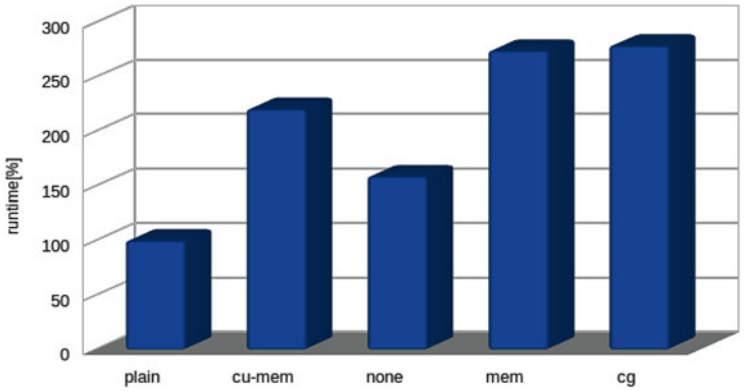


Fig. 6.3 Relative runtime of just the CMD code ('plain') compared to running it with cuda-memcheck ('cu-mem'), the Valgrind none-tool ('none'), Memcheck ('mem') and Cudagrind ('cg')

the latter is not handled by Valgrind. To illustrate this effect a CUDA-based MD code, developed at HLRS, is being used.

This code has been executed on the HLRS' Sandy Bridge based 'Laki' cluster.[4] The nodes contain E5-2670 Sandy Bridge processors, 12 GB of RAM and a GeForce GTX 680 accelerator. 72 % of the runtime is spent on kernel execution, 28 % on host code, including driver API calls. This version of the code features almost no overlapping of host and device code.

Figure 6.3 shows the relative, average runtime without any tools (**plain**, baseline 100 %), with `cuda-memcheck` (**cu-mem**), the Valgrind 'none' tool (**none**), the Valgrind Memcheck tool (**mem**) and finally Cudagrind (**cg**). Note that running Cudagrind implies using the Valgrind Memcheck tool. The 'none' tool shows the basic overhead of running a program with Valgrind, as it does not perform any nor changes the code being executed. Each variant has been run 20 times with the variation between runs being less than 5 %.

As Fig. 6.3 shows, the overhead of using Cudagrind over checking a program with Valgrind's Memcheck tool is less than 3 %. The overhead of Memcheck itself is 175 % and for simply running the tool through Valgrind, without performing any checks, 60 %. For NVIDIA's cuda-memcheck, which only works on the accelerator side, it is 120 %.

[4] https://www.hlrs.de/systems/platforms/nec-cluster-laki-laki2/

6.7 Summary

Cudagrind offers a novel approach to close the missing link between host side memory checking, e.g. through Valgrind based tools, and device side memory checking with `cuda-memcheck`. The provided set of wrappers allow runtime tracking of allocation status and definedness of memory during transactions between host and/or device. All without the need for recompilation, except for the exact location of errors in the source code, and depending only on the existence of a valid Valgrind installation and the CUDA driver API.

With Cudagrind operating on the CUDA driver level it works with 'everything CUDA'. This includes programs relying on high level constructs like CUDA enabled libraries and pragma based programming models, e.g. OpenACC. But also the still very common method of directly utilizing the CUDA runtime API and even the low level approach of accessing the device directly through the CUDA driver or it's respective API.

Section 6.6 has shown that there is virtually no overhead when running Cudagrind on top of Valgrind's Memcheck tool. Additionally, due to the way Valgrind dynamically loads wrappers at runtime, there is no overhead when running the program on it's own. The original program does not need to changed or recompiled/-linked in any way.

At the time of writing a beta Version of Cudagrind has been released.[5]

6.7.1 Outlook

The work presented in this paper offers a solid foundation for memory transaction checking on CUDA enabled programs.Future versions might incorporate additional wrappers that check the arguments of kernels that are started on the device. One way to achieve this is a wrapper for `cuLaunchKernel` that unpacks the parameter list passed to the function in order to retrieve and check the variables passed to the kernel.

Another possible approach would be a thorough integration of CUDA into the Valgrind framework. Akin to how Valgrind provides a virtual x86 CPU, to execute arbitrary binary programs, it would be possible to extend it's core with a virtual device able to execute PTX or CUBIN code. This would allow handling of code produced by the NVIDIA CUDA compiler as well as third party products. This would allow full control and complete memory checking capabilities. Alas, it's no trivial task. One possible way to achieve this goal lies in the extension and integration of a dynamic CUDA compilation frameworks like 'GPU Ocelot' [1]. These provide a runtime promising efficient execution of PTX and CUBIN code

[5] https://www.hlrs.de/organization/av/spmt/research/cudagrind/

on multi-core architectures. A reversed approach, as proposed in [2], would be to include the instrumentation in Ocelot itself. But in this case the host side memory checking capabilities, offered by Valgrind, would have to be integrated into GPU Ocelot.

Acknowledgements This work was supported by the H4H project funded by the German Federal Ministry for Education and Research (grant number 01IS10036B) within the ITEA2 framework (grant number 09011).

References

1. Diamos, G.F., Kerr, A.R., Yalamanchili, S., Clark, N.: Ocelot: a dynamic optimization framework for Bulk-synchronous applications in heterogeneous systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, Vienna, pp. 353–364. ACM, New York (2010). http://doi.acm.org/10.1145/1854273.1854318
2. Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S., Schwan, K.: A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, Newport Beach, pp. 9:1–9:9. ACM, New York (2011). http://doi.acm.org/10.1145/1964179.1964192
3. Johnson, S.C.: Lint: A C Program Checker. Computing Science Technical Report 65, Bell Laboratories, (1977)
4. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007). http://doi.acm.org/10.1145/1273442.1250746
5. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, Anaheim, pp. 17–30. USENIX Association, Berkeley (2005)

Chapter 7
Node Performance and Energy Analysis with the Sniper Multi-core Simulator

Trevor E. Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout

Abstract Two major trends in high-performance computing, namely, larger numbers of cores and the growing size of on-chip cache memory, are creating significant challenges for evaluating the design space of future processor architectures. Fast and scalable simulations are therefore needed to allow for sufficient exploration of large multi-core systems within a limited simulation time budget. By bringing together accurate high-abstraction analytical models with fast parallel simulation, architects can trade off accuracy with simulation speed to allow for longer application runs, covering a larger portion of the hardware design space. Sniper provides this balance allowing long-running simulations to be modeled much faster than with detailed cycle-accurate simulation, while still providing the detail necessary to observe core-uncore interactions across the entire system. With per-function advanced visualization and coupled power and energy simulations, the Sniper multi-core simulator can provide a fast and accurate way both to understand and optimize software for current and future hardware systems.

7.1 Sniper Overview

Sniper is a next generation parallel, high-speed and accurate x86 simulator. This multi-core simulator is based on the interval core model and the Graphite simulation infrastructure [11], allowing for fast and accurate simulation and for trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures.

The Sniper simulator allows one to perform timing simulations for both multi-program workloads and multi-threaded, shared-memory applications running on 10s to 100+ cores, at a high speed when compared to existing simulators. The main

T.E. Carlson (✉) • K. Van Craeynest • L. Eeckhout
Ghent University, Gent, Belgium
e-mail: trevor.carlson@elis.ugent.be

W. Heirman
Intel, ExaScience Lab, Leuven, Belgium

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_7

feature of the simulator is its core model which is based on interval simulation, a fast mechanistic core model [4]. Interval simulation raises the level of abstraction in architectural simulation which allows for faster simulator development and evaluation times; it does so by 'jumping' between miss events, called intervals [8]. Sniper has been validated against multi-socket Intel Core 2 systems and provides average performance prediction errors within 25 % at a simulation speed of up to several MIPS [2].

This simulator, and the interval core model, is useful for uncore and system-level studies that require more detail than the typical high-level models, but for which cycle-accurate simulators are too slow to allow workloads of meaningful sizes to be simulated. As an added benefit, the interval core model allows the generation of CPI stacks, which show the number of cycles lost due to different characteristics of the system, like the cache hierarchy or branch predictor, and lead to a better understanding of each component's effect on total system performance. This extends the use for Sniper to application characterization and hardware/software co-design. The Sniper simulator is available for download at http://snipersim.org and can be used freely for academic research.

7.2 Speeding Up Simulation

In this section we describe a number of strategies that, when used together, provide the methods where we speed up micro-architectural simulation in Sniper while still maintaining accuracy.

7.2.1 Interval Simulation

Interval simulation is a recently proposed simulation approach for simulating multi-core and multiprocessor systems at a higher level of abstraction compared to current practice of detailed cycle-accurate simulation [8]. Interval simulation leverages a mechanistic analytical model to abstract core performance by driving the timing simulation of an individual core without the detailed tracking of individual instructions through the core's pipeline stages. The foundation of the model is that miss events (branch mispredictions, cache and TLB misses) divide the smooth streaming of instructions through the pipeline into so called intervals [4]. The branch predictor and cache models are simulated in detail. The cache hierarchy, cache coherence and interconnection network simulators determine the miss events; the analytical model derives the timing for each interval. The cooperation between the mechanistic analytical model and the miss event simulators enables the modeling of the tight performance entanglement between co-executing threads on multi-core processors.

The multi-core interval simulator models the timing for the individual cores. The simulator maintains a 'window' of instructions for each simulated core. This window of instructions corresponds to the reorder buffer of a superscalar out-of-order processor, and is used to determine miss events that are overlapped by long-latency load misses. The functional simulator feeds instructions into this window at the window tail. Core-level progress (i.e., timing simulation) is derived by considering the instruction at the window head. In case of an I-cache miss, the core simulated time is increased by the miss latency. In case of a branch misprediction, the branch resolution time plus the front-end pipeline depth is added to the core simulated time, i.e., this is to model the penalty for executing the chain of dependent instructions leading to the mispredicted branch plus the number of cycles needed to refill the front-end pipeline. In case of a long-latency load (i.e., a last-level cache miss or cache coherence miss), we add the miss latency to the core simulated time, and we scan the window for independent miss events (cache misses and branch mispredictions) that are overlapped by the long-latency load—second-order effects. For a serializing instruction, we add the window drain time to the simulated core time. If none of the above cases applies, we dispatch instructions at the effective dispatch rate, which takes into account inter-instruction dependencies as well as their execution latencies. We refer to [8] for a more elaborate description of the interval simulation paradigm.

7.2.2 Parallel Simulation

An additional way to speed up architectural simulation in the multi/many-core era is to parallelize the simulation infrastructure in order to take advantage of increasing core counts. This is needed because most of the improvement in processor performance recently has come from increasing the number of cores per processor, and therefore parallelizing the simulator allows us to take advantage of this performance trend.

One of the key issues in parallel simulation is the balance of accuracy versus speed. Cycle-by-cycle simulation advances one cycle at a time, and thus a parallel simulator's threads (which map directly to the application threads) would need to synchronize every cycle to maintain accuracy. The resulting performance of this approach is not very high because it requires barrier synchronization between all simulation threads at every simulated cycle. If the number of simulator instructions per simulated cycle is low, parallel cycle-by-cycle simulation is not going to yield substantial simulation speed benefits and scalability will be poor.

There exist a number of approaches to relax the synchronization imposed by cycle-by-cycle simulation [7]. A popular and effective approach is based on barrier synchronization. The entire simulation is divided into quanta, and each quantum comprises multiple simulated cycles. Quanta are separated through barrier synchronization. Simulation threads can advance independently from each other between barriers, and simulated events become visible to all threads at each barrier.

The size of a quantum is determined such that it is smaller than the critical latency, or the time it takes to propagate data values between cores. Barrier-based synchronization is a well-researched approach, see for example [13].

More recently, researchers have been trying to relax multi-threaded simulation even further, beyond the critical latency. If done correctly, the idea would be to gain a simulation speed while accepting a small amount of simulation error. When taken to the extreme, no synchronization is performed at all, and all simulated cores progress at a rate determined by their relative simulation speed. This will introduce skew, or a cycle count difference between two target cores in the simulation. This in turn can cause causality errors when a core sees the effects of something that—according to its own simulated time—did not yet happen. These causality errors can be corrected through techniques such as checkpoint/restart, but they are allowed to occur and are accepted as a source of simulator inaccuracy. Chen et al. [3] study both unbounded slack and bounded slack schemes; Miller et al. [11] study similar approaches. Unbounded slack implies that the skew can be as large as the entire simulated execution time. Bounded slack limits the slack to a preset number of cycles, without incurring barrier synchronization.

In the Graphite simulator [11], a relaxed synchronization scheme has been adopted, trading off causality errors in the simulation for the ability to speed up the simulator even further. The simulation infrastructure presents a number of different synchronization strategies. The barrier method provides the most basic synchronization, requiring cores to synchronize after a specific time interval, as in quantum-based synchronization. The most loose synchronization method in Graphite (lax synchronization) does not synchronize at all except when explicitly performed by the application. The random-pairs synchronization method is in the middle between these two extremes and is implemented by random choosing two simulated target cores to synchronize. If the two target cores are out of sync, the simulator stalls the core that runs ahead waiting for the slowest core to catch up.

We evaluated these synchronization schemes in terms of accuracy and simulation speed [2] and have determined that lax synchronization can result in a very high simulation error. In Sniper, we therefore use barrier synchronization with a 100 ns quantum to minimize error.

7.3 Sniper Components

Sniper [2] is the combination of three main components. In Fig. 7.1, we provide an illustration of how these pieces fit together (Note that the simulation infrastructure itself is not shown). The first is the functional execution model which provides the dynamic instruction stream of the running application. It is this stream of instructions that Sniper uses to determine the timing effects of a particular microarchitecture.

The second component is the timing simulation component. The timing models interact with the dynamic instruction stream to determine the rate of forward

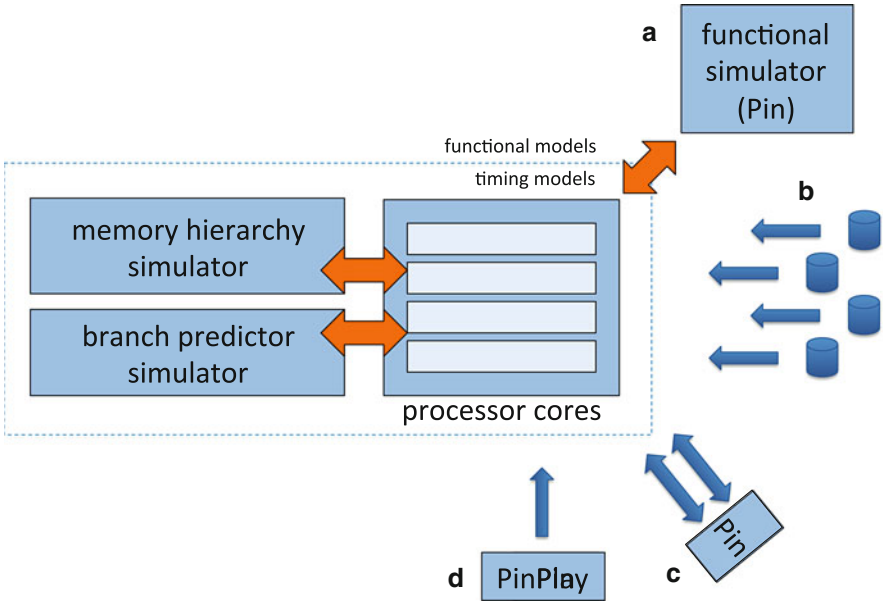


Fig. 7.1 A high-level diagram of the functional and timing components used in Sniper. There are a number of different options to connect an instruction trace to Sniper such as (**a**) compiling Sniper as a pintool, (**b**) using stored SIFT traces in a single- or multi-program configuration, (**c**) using a number of single- or multi-threaded Pin program with the dynamic bidirectional SIFT protocol, or (**d**) using PinPlay to replay a number of application pinballs. When using (**a**), the core and memory timing models are integrated into a single Pin tool. When using (**b**), (**c**) or (**d**), the timing models operate in their own process, while the Pin tool attaches to the application directly. *Orange arrows* represent communication inside of a process, while *blue arrows* are bidirectional UNIX pipes that connect the Pin process to the Sniper timing process

progress that is being made. The Sniper Multi-Core Simulator brings together the interval core timing model (as outlined in Sect. 7.2.1) with a number of detailed microarchitecture models, such as the Pentium M (Dothan) style branch predictor [14], and instruction and data caches, to model timing in the core and memory subsystems. The data cache models support a large number of configuration options, such as private, and shared last-level caches, multiple DRAM controllers (to support NUMA architectures) and hardware prefetchers. Some timing models are simulated at a higher level of abstraction to speed up simulation. The interconnects between the cache components are modeled as queues, with the total latency consisting of two parts: a minimum access latency and a queuing delay that increases as the link's bandwidth is consumed.

The third component in the Sniper multi-core simulator is the event simulation infrastructure that keeps track of events in the system and keeps them in sync with one another. This component is largely based on the Graphite simulator [11]. For more details on the parallel simulation aspect of Sniper, see Sect. 7.2.2.

The functional front-end provides a dynamic instruction stream of instructions to the timing models of the simulator. However, Sniper is not a functional-first simulator, but most closely resembles a functional-directed simulator with timing-feedback [1], where the timing models can cause back-pressure on the functional models to control the rate of progress.

Traditionally, Sniper was used as a standalone Pin [10] tool. One drawback of these earlier versions of Sniper was that it was restricted to simulating just a single application at a time, preventing Sniper from running multi-program or multiple multi-threaded workloads. To address this issue, we developed the SIFT protocol. SIFT, the Sniper instruction trace format, is an optionally compressed binary format that contains instruction information along with dynamic execution results of the instruction execution (branch taken information, conditionally executed instruction information, and other details). The SIFT protocol can serve as either a file format for single-threaded traces or as a bi-directional communication channel between the functional trace generator and the timing simulator.

As an alternative to stored SIFT traces, Sniper also supports replaying single-threaded PinPlay [12] application snapshots (pinballs) via the SIFT protocol. Pinballs are deterministic execution snapshots where dynamic execution traces are replayed by executing the application directly instead of storing a dynamic trace of the running application. All communication with the operation system (such as file I/O and system calls) and the results of relevant instructions (such as CPUID and RDTSC) are captured and played back to the application to enforce deterministic application replay. One added benefit when using pinballs as an application file trace is the small file sizes that can be produced. The size of the pinballs generated are related to the size of the application binary and input data instead of the total number of dynamic instructions that are executed in the trace. This can significantly reduce the size of the files stored.

Apart from single-threaded traces, Sniper supports multiple single-threaded applications (from any combination of inputs, either stored SIFT traces, pinballs or single-threaded applications), as well as more complex configurations, such as multi-threaded applications and multiple processes that can consist of multiple threads themselves. In addition, we support MPI applications that use shared-memory to communicate, and we do this through virtual-to-physical mapping of the native application addresses. Additionally, applications that use OpenMP inside MPI ranks are also supported.

To support these configurations, especially for multi-threaded applications, a bi-directional SIFT channel is required as the channel enforces timing back-pressure making sure that the functional execution of each thread across all processes are kept in sync relative to one another. The SIFT channels are also used to communicate events such as thread spawning, important system calls (such as OS/pthread synchronization requests) and other out-of-band requests, such as reading or writing to memory.

7.4 Software Analysis and Visualization

To aid in hardware/software co-optimization we have developed application- and hardware-level software analysis extensions in Sniper to help the application designers better understand the interactions of their software with modern or future hardware designs.

Through an extension of interval analysis [6], the CPI stack for out-of-order cores [5] was developed to provide a new level of insight into the causes of performance loss when running an application in an out-of-order processor. The CPI stack shows the cycles lost due to different events in the system, such as LLC cache misses or branch misprediction penalties. As described in Sect. 7.2.1, each stall event triggers a corresponding penalty from the interval simulation model, which corresponds directly to each of the CPI stack components. The base CPI component measures the obtainable performance according to the amount of ILP that can be extracted from the application by the reorder buffer.

A straight-forward extension to the single-threaded CPI stack is a multi-threaded version, normalized by percent of run time, where the causes of slowdown for each thread can be seen easily. In the example in Fig. 7.2, we present both a homogeneous application along with a homogeneous microarchitecture. Though the use of CPI stacks, we can determine that the longer run time of the first processor is caused by waiting for a barrier to return. The barrier waiting time is about equal to the extra time that the second processor needs to access the memory hierarchy on the other processor. Changing this initial allocation policy, therefore, could reduce the off-socket latency and improve the performance of the application. CPI stacks allows a software developer to directly determine the causes of performance loss and therefore helps the developer to choose where to optimize the software next.

While applications with a few cores are relatively easy to reason about in this form of multi-threaded CPI stack, it becomes more difficult to understand how

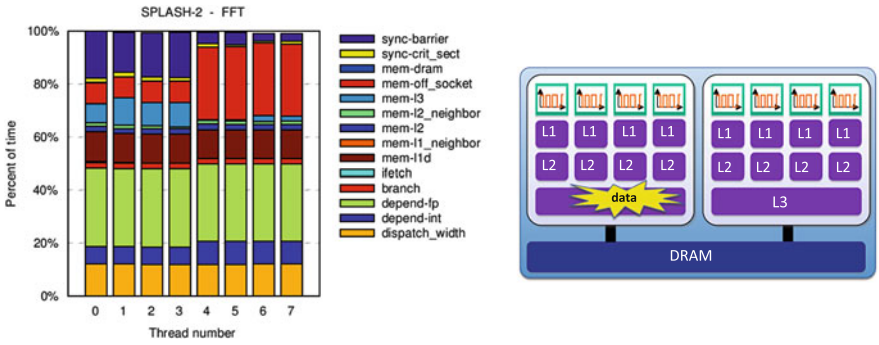


Fig. 7.2 A CPI stack of the homogeneous multi-threaded application, SPLASH-2, FFT (*left*) on a homogeneous architecture (*right*). Because of NUMA effects, where the initial data has been allocated to the L3 of the first processor, threads from another processor need to wait for the data to be transferred from the remote socket

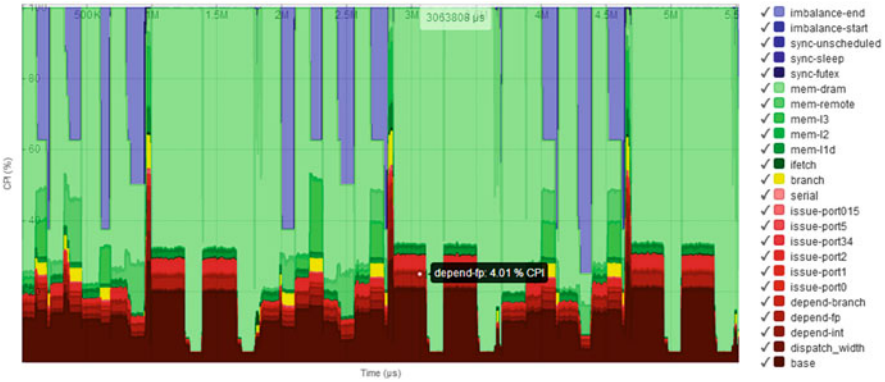


Fig. 7.3 Sniper visualization output of the CPI stack components of an application in detail. The website view is dynamic (one can zoom into regions of interest) and automatically generated from the Sniper simulation

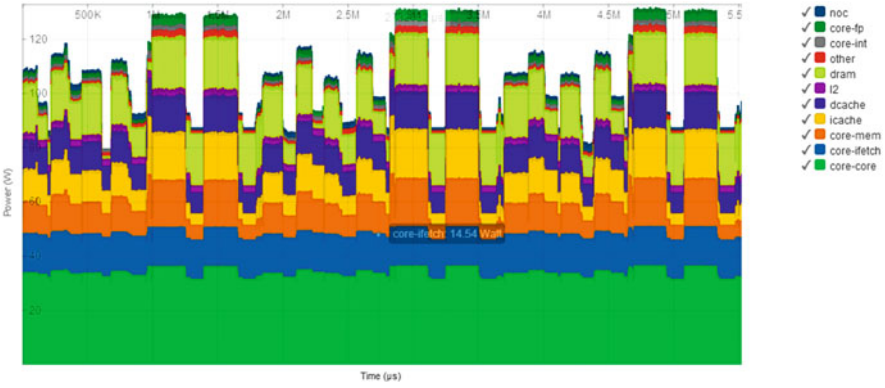


Fig. 7.4 Sniper visualization output of the power stack components of an application in detail

hundreds or more CPI stacks come together to contribute to overall run time. A more insightful way to view this data is to view how the CPI stacks change over time, not for a single thread, but for all threads in aggregate. In this way, we can better understand how the phase behavior affects the progress of the application for a given time slice.

With the visualization features recently added to version 4.1 of Sniper, this ability to view how the aggregate CPI stacks for the entire system changes over time was added. Examples of these features are shown in Figs. 7.3 and 7.4 which shows an example CPI stack from an HPC application, and the corresponding power breakdown over time for each component in the system, respectively. Each simulation can now be augmented with an automatically generated website that allows the user to dynamically change the view of the data at hand: by zooming and providing more or less detail as needed. In addition to CPI stack data, power stacks

can also be created using this aggregate view to provide similar insights. Since version 3.2 of Sniper, we have integrated McPAT and demonstrated that accurate power and energy numbers can be obtained with higher-level core models [9].

Although CPI stack visualization provides the necessary data to broadly optimize a particular benchmark, knowing where to look inside of a benchmark to find and correct the problem can be a bit more difficult. Most programmers would prefer to understand the breakdown of each CPI component on a per-function basis instead of being broken-down on a per time quantum basis. As a potential solution, we integrated per-function statistics that go beyond what a typical performance analysis tool can provide, even when using performance counters. Examples of data items that can now be exposed are individual CPI stack components on a per-function basis. Additional data that can be more useful in bandwidth-constrained environments include monitoring cache-line use efficiency.

One recently developed application visualization method is called the Roofline model [15]. The Roofline model represents the intersection of two lines, the maximum bandwidth that can be exposed by a given machine, as well as the peak floating-point performance of that machine. By plotting applications on the Roofline model, one can determine how close your application is to the maximum available bandwidth and computational resources. These two resources tend to be the largest limiting factors for many compute-intensive applications like those found in HPC-style workloads. In Fig. 7.5 we show an example of an automatically generated Roofline model in Sniper. In the Roofline plot, each function of an application under study can be easily evaluated to measure how it performs compared to the maximum performance of a given simulated microarchitecture.

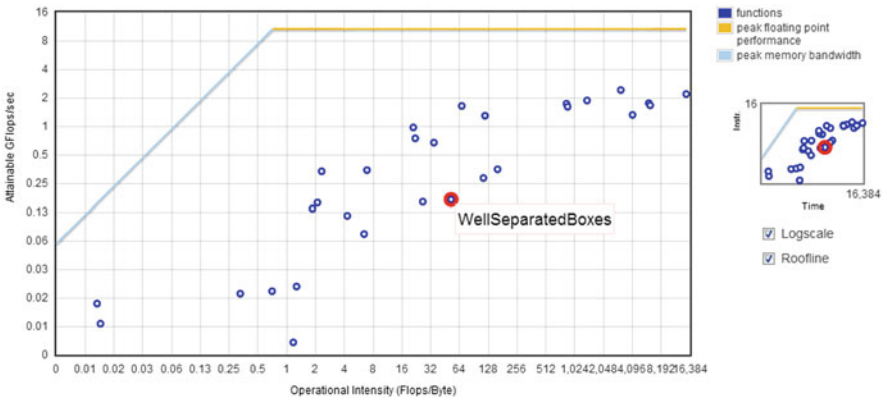


Fig. 7.5 Sniper visualization output of the Roofline model. Using the theoretical maximum bandwidth and floating-point operations per second, we can compare the performance of each function to determine how close we are to the theoretical maximum performance of that function

7.5 Accuracy and Validation

When viewing the results of a software application simulation run or hardware update to improve performance, it is important to understand how well the underlying tools compare to current, state of the art technologies that they are attempting to model. When developing the Sniper simulator, we wanted to focus on accuracy through white-box modeling of the different components of the system. We therefore used the mechanistic interval simulation technique which allows one to configure the system based on micro-architectural parameters. This provides the benefit of being able to update the system configuration for potential future design points. We then combined this model along with the other essential components of a modern processor to form the Sniper Multi-Core Simulator.

Sniper has been validated against real hardware models. In a recent paper [2] we have demonstrated good accuracy compared to real hardware. On the Splash-2 benchmarks [16], we achieve an average absolute error of 25 %, with relative scaling numbers performing even better, tracking the hardware results closely. In addition, we demonstrated the accuracy of Sniper combined with McPAT through validation against real hardware; we reported average power prediction errors of 8.3 %, for a set of SPEC OMP2001 benchmarks [9].

7.6 Conclusion

The increasing complexity of software (with multi-threaded applications and ever-increasing data set sizes) coupled with increasingly complex microarchitectures (with 60+ cores in the Xeon Phi and larger LLC sizes), simulation of new software on next generation hardware is becoming increasingly difficult to perform in a timely manner. Coupling traditional performance simulations with energy and power simulations only exacerbates the problems moving forward.

By providing a detailed understanding of both the hardware and software, and allowing for a number of accuracy and simulation performance trade-offs, we show that Sniper can be a useful complement to traditional performance analysis tools for evaluating and optimizing software for high-performance multi-core and many-core systems.

Acknowledgements We thank Mathijs Rogiers for his invaluable work on the visualization features of Sniper and the anonymous reviewers for their valuable feedback. This work is supported by Intel and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC Grant agreement no. 259295. Experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium; the Intel HPC Lab, Swindon, UK; and the VSC Flemish Supercomputer Center.

References

1. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., Ortega, D.: COTSon: infrastructure for Full System Simulation. ACM SIGOPS Oper. Syst. Rev. **43**(1), 52–61 (2009)
2. Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Seattle, pp. 52:1–52:12 (Nov 2011)
3. Chen, J., Dabbiru, L.K., Wong, D., Annavaram, M., Dubois, M.: Adaptive and speculative slack simulations of CMPs on CMPs. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Atlanta, pp. 523–534 (Dec 2010)
4. Eyerman, S., Eeckhout, L., Karkhanis, T., Smith, J.E.: A mechanistic performance model for superscalar out-of-order processors. ACM Trans. Comput. Syst. (TOCS) **27**(2), 42–53 (2009)
5. Eyerman, S., Eeckhout, L., Karkhanis, T., Smith, J.: A top-down approach to architecting CPI component performance counters. Micro, IEEE **27**(1), 84–93 (2007)
6. Eyerman, S., Smith, J., Eeckhout, L.: Characterizing the branch misprediction penalty. In: Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, pp. 48–58 (Apr 2006)
7. Fujimoto, R.M.: Parallel discrete event simulation. Commun. ACM **33**(10), 30–53 (1990)
8. Genbrugge, D., Eyerman, S., Eeckhout, L.: Interval simulation: raising the level of abstraction in architectural simulation. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, pp. 307–318 (Feb 2010)
9. Heirman, W., Sarkar, S., Carlson, T.E., Hur, I., Eeckhout, L.: Power-aware multi-core simulation for early design stage hardware/software co-optimization. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, pp. 3–12 (Sept 2012)
10. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, pp. 190–200 (June 2005)
11. Miller, J.E., Kasture, H., Kurian, G., Gruenwald III, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A.: Graphite: a distributed parallel simulator for multicores. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, pp. 1–12 (Jan 2010)
12. Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J.: PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Toronto, pp. 2–11 (Apr 2010)
13. Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., Wood, D.A.: The Wisconsin wind tunnel: virtual prototyping of parallel computers. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Clara, pp. 48–60 (May 1993)
14. Uzelac, V., Milenkovic, A.: Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, pp. 207–217 (Apr 2009)
15. Williams, S., Waterman, A., Patterson, D.A.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (Apr 2009)
16. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22th International Symposium on Computer Architecture (ISCA), Portofino, pp. 24–36 (June 1995)

Chapter 8
A Comparison of Trace Compression Methods for Massively Parallel Applications in Context of the SIOX Project

Alvaro Aguilera, Holger Mickler, Julian Kunkel, Michaela Zimmer, Marc Wiedemann, and Ralph Müller-Pfefferkorn

Abstract The analysis and optimization of HPC I/O is a daunting task that is still unaddressed at large. The SIOX project aims to help HPC users and system administrators alike to improve the I/O performance of the applications by gaining awareness of the I/O operations taking place on the system and launching corrective measures when a problem is encountered. Given the size of modern HPC clusters and the corresponding amount of I/O they generate, the SIOX project faces a series of scalability challenges that need to be resolved. Beyond presenting the architecture and functioning of the SIOX system, this paper examines one of its biggest challenges, namely the transmission and management of large amount of event-based I/O trace information as well as the benefits the use of the trace compression techniques like ScalaTrace and C3G may convey.

8.1 Introduction

The main objective of the Scalable I/O for Extreme Performance (SIOX) project [9] is to obtain an overview of all I/O calls occurring on a given High Performance Computing (HPC) system, and use this information to conduct run time optimizations of the I/O subsystem. This requires among other things, the continuous and undelayed transmission of trace data from the SIOX clients to the SIOX servers. One of the biggest challenges the project faces to ensure the scalability of its architecture, is to adequately reduce the size of this data flow, allowing the SIOX servers to keep pace with large numbers of clients sending them data. If we consider the fact that

A. Aguilera (✉) • H. Mickler • R. Müller-Pfefferkorn
Technische Universität Dresden, D-01062 Dresden, Germany
e-mail: alvaro.aguilera@tu-dresden.de

J. Kunkel
Deutsches Klimarechenzentrum, D-20146 Hamburg, Germany

M. Zimmer • M. Wiedemann
Universität Hamburg, D-20148 Hamburg, Germany

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_8

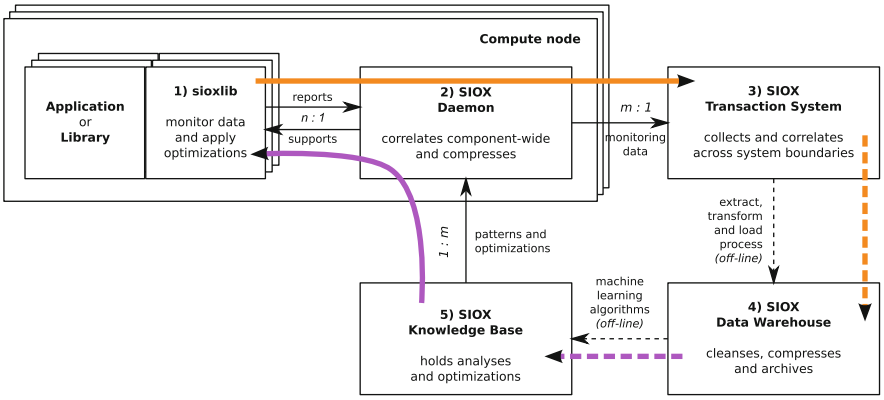


Fig. 8.1 SIOX's data collection and optimization workflow

most HPC applications follow the Single Program, Multiple Data (SPMD) paradigm [7] and that because of this the I/O operations are mostly symmetrical among the nodes executing a parallel application, it is reasonable to expect that the use of compression algorithms at appropriately chosen points of SIOX's data collection and optimization workflow (see Fig. 8.1) can significantly reduce the amount of data that needs to be transmitted to the servers without losing information content.

In this paper we begin by offering a general overview of the SIOX architecture in Sect. 8.2. This is followed in Sect. 8.3 by introducing the problem of I/O tracing by means of a concrete example at the German Climate Computing Center (DKRZ). In Sect. 8.4, we characterize the type of tracing information that is relevant for SIOX. This is done by examining public trace data originated from scientific applications during production use. In Sect. 8.5, we present the functioning of two specialized compression methods: ScalaTrace and the C3G library. These two methods are used to compress the previously presented trace data and the results are presented in Sect. 8.6. We conclude the paper in Sect. 8.7 by giving our thoughts about the usability of these trace data compression methods in SIOX as well as by hinting at further research in this topic.

8.2 SIOX Architecture

Let's dive straight into the architecture and workflow of the SIOX system, which is depicted in Fig. 8.1. The whole system consists of five main software parts: the monitoring library, the daemons, the transaction system, the data warehouse, and the knowledge base. It is designed to allow the system to propose optimization calls at runtime using an automatic, iterative improvement procedure.

At first, SIOX needs some data to work with. This data is gathered both by means of the sioxlib and SIOX daemons that run on every node taking part in

carrying out I/O operations. While a SIOX daemon collects statistics-like data (e.g. network throughput, CPU utilization), the applications or libraries augmented by sioxlib yield I/O events (e.g. "PID 4857 starts POSIX read() operation on fd 5 at 1373028159.784392937UTC"). Augmenting arbitrary programs is realized by using wrapper libraries which are interposed between the program and the original library that contains the actual functionality (e.g. `read()` inside libc). The interpositioning can be realized at link-time (using ld's *wrap* capabilities) or at run-time using library preloading. SIOX contains a wrapper generator which can be used to facilitate creating wrapper libraries for both flavors.

All data from instrumented programs is funneled into the node-local SIOX daemon which is responsible for forwarding data to the transaction system where it is stored for further analysis.

The data warehouse holds consolidated information that is produced by extract, transform, and load (ETL) processes from the data in the transaction system.

The knowledge base contains the knowledge about aggregated information of the data warehouse, the hardware topology (i.e. network connections) and characteristics, and the parameters for optimization (e.g. "A drop in I/O throughput was observed for packets smaller than 4 MiB; therefore aggregation of smaller packets to 4 MiB packets is proposed by the SIOX system").

8.2.1 Inner Structure

Now that the main components of the SIOX system have been described, we can take a closer look on how the augmented processes interact with their daemon. The data captured by sioxlib will be assembled into so-called *activities* and handed over to an activity multiplexer (AMux, see Fig. 8.2). The AMux as well as almost all inner parts are laid out as plug-ins so that it is easy to extend the system.

When receiving activities from sioxlib, the AMux feeds them to all plug-ins that have subscribed. Here, the activity forwarding plug-in must always be present – the AMux of the node-local SIOX daemon subscribes to it for further processing; at this point being aware of all the activities of the other augmented processes on this node. Additionally, the SIOX daemon is equipped with a statistics multiplexer (SMux) which handles the aforementioned node performance data.

Each SIOX daemon itself has an AMux with activity forwarder, which makes it possible to create a cascading hierarchy of daemons. This helps reducing the pressure on the transaction system, e.g. by compressing the oftentimes very similar data from different nodes.

The reasoner module finally has a twofold task: At runtime, it receives the reports of the various anomaly detection plug-ins within its scope (process, node or system-wide) and compiles an assessment of every subsystem's health status from them. Neighbouring reasoners will communicate periodically to relate local to global health, and use this data to adjust SIOX's monitoring (e.g., throttle granularity at the forwarders or deactivate analysis plug-ins) or issue warnings to the administrator.

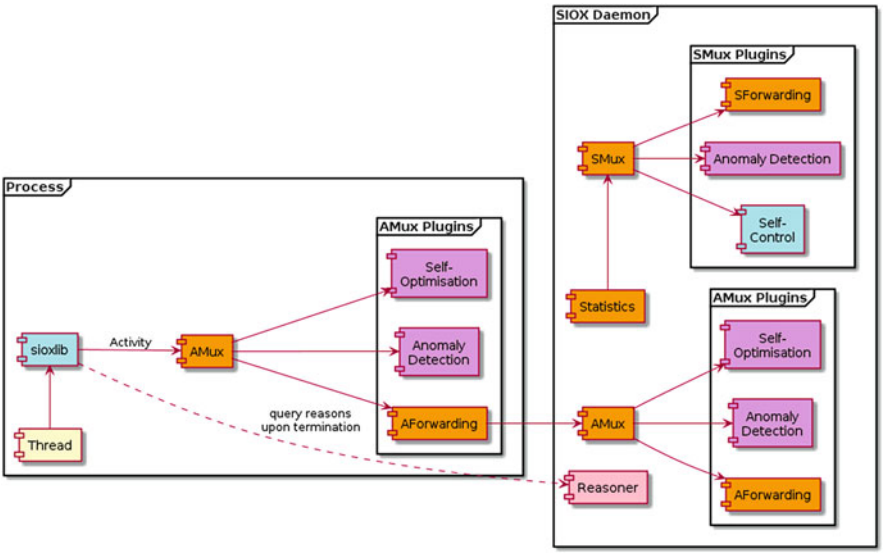


Fig. 8.2 Interactions between process and daemon plug-ins

Upon job completion, the reasoner will compile a human-readable performance report from the data gathered, indicating problems and possible improvements to the user.

8.3 The Data Deluge of Constant Monitoring

The SIOX system aims to monitor the parallel I/O in the data center on a 24/7 basis, extracting knowledge through the use of machine learning techniques, and ultimately using it to optimize I/O. The monitoring and knowledge cycle is illustrated in Fig. 8.1 and further described in [9]. Unfortunately, constant monitoring leads to an unbearable volume of data. Let's exemplify this issue for the Blizzard supercomputer hosted by the DKRZ. Delivering a maximum throughput of 30 GB/s on its 9 PB storage, it consists of 12 I/O servers controlling about 7,000 hard disks. Assuming a program writes 1 GB of data to a file; if we monitor all block accesses on the hard disks in a granularity of 4 KB blocks, this would lead to roughly 250,000 events. If each monitored event has a space requirement of about 20 bytes, then about 5 MB of data must be recorded – corresponding to an overhead of 0.5 %.

While this volume seems insignificant and bearable, it accounts neither for the subsequent reading or re-writing of data, nor for the multitude of software layers involved in the operations. This analysis would look different if we begin with the observed I/O rate: The I/O servers at DKRZ process in average about 10 GB/s of data, thus an additional stream of 50 MB/s would be needed to store the trace data.

Over a year this would accumulate to about 1.6 PB or 17 % of the total storage. One might argue that nobody should record individual block accesses on disk but just higher-level I/O. Unfortunately, mainly small files are stored on the file system and many applications also access data in small chunks. Therefore, we consider the space overhead of 0.5 % a conservative estimate when monitoring I/O across software layers on clients and servers.

8.4 Characterization of I/O Traces

In general, the I/O trace logs to be sent to the SIOX servers are event-based and have the form of a list in which the logged I/O functions of different instrumented layers appear together with the parameters that were used to call them. The format of this list must preserve the causal relations of the entries, so that for instance the parent-child correspondence between different function calls can be reconstructed. Additionally, a timestamp is added to each logged function call to allow a subsequent temporal correlation, and since a spatial correlation is also required, context information such as node-id, process number, user-id, etc. is also transmitted along the trace entries.

Parallel applications running on HPC systems normally exhibit three distinct phases of I/O: an *input* phase when the application is started and it reads the data required to do the computations, the periodically occurring writing of *checkpointing* data to resume the application in case its execution is aborted, and finally the *output* phase in which the results are written to disk. Some classes of applications also periodically flush their internal state for later inspection. Examples of this class are climate applications, which record the temporal and spatial changes of oceanic, atmospheric and land climate over decade frequencies. This also involves particle transfer from one layer the other, varying the I/O load. The state flushing behavior is quite similar to regular checkpointing and thus needs no special treatment.

Even though the characteristics of an I/O trace will obviously depend on the application producing it, as well as on its use case, we can examine previously published I/O traces produced during real runs of scientific applications to get an idea of the magnitude and features to expect in the traces to be produced by SIOX. One such example are the I/O traces published in 2009 by the Scalable I/O project at Sandia National Laboratories [8]. These trace files are listed in Table 8.1 and they contain the I/O calls that were realized during production runs of the parallel applications `Alegra`, `CTH` and `S3D`.

The rest of this section briefly describes the general features exhibited by these traces. A more detailed analysis of them is given by their creators in [5].

Table 8.1 I/O trace files

Application	Description	Cores	Running time	Size (MB)
Alegra	Shock simulator	2,744	10 min 15 s	169
Alegra		5,830	15 min 43 s	352
CTH	Shock simulator	3,360	3 h 29 min 23 s	829
S3D	Navier-Stokes solver	6,400	7 min 51 s	210
S3D		6,400	14 min 35 s	99
S3D		1,024	5 min 17 s	15

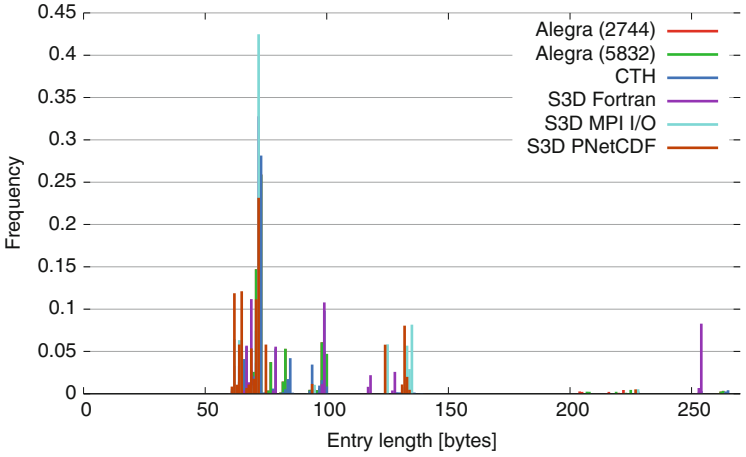


Fig. 8.3 Entry lengths found in the I/O traces from Sandia Labs

8.4.1 Average Entry Length and Their Temporal Distribution

Insight about the expected average and maximal size of the entries composing the I/O traces is important when dimensioning the SIOX system since they help us make a first estimation about the bandwidth that will be required to transport the tracing information to the servers as well as the space needed for its storage. A histogram showing the different entry sizes inside the analyzed traces is shown in Fig. 8.3. The size of individual entries in these traces varied between 57 and 265 bytes with an arithmetic mean of 83 bytes. The biggest entries corresponded to the `open()` function calls due to the ASCII paths contained in them.

Now that a plausible size range for the trace events has been determined, we are interested in estimating the rate at which they will arrive at the SIOX servers. The temporal distribution of the I/O calls is characterized by intervals of intense activity corresponding to one of the three I/O phases we've already mentioned, followed by some idle time in which the nodes are busy computing or communicating. This behavior can be clearly seen in Fig. 8.4, where the number of I/O calls occurring in the traces is visualized over time. Compared to others, Alegra performs most

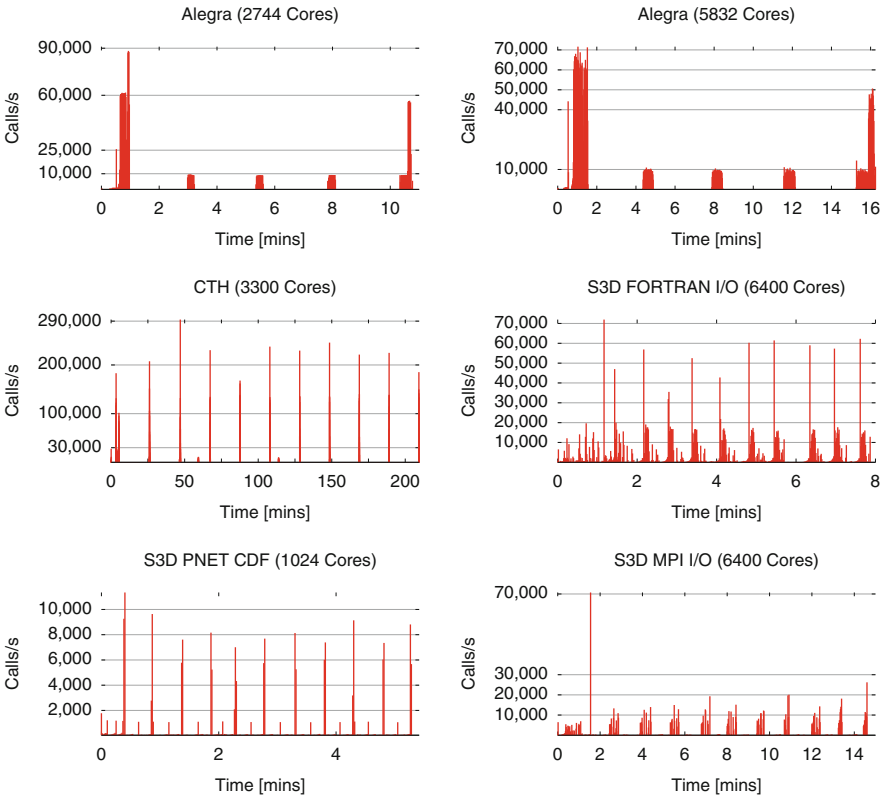


Fig. 8.4 Temporal distribution of the I/O calls

Table 8.2 Estimation of the network load due to trace information

Application	Cores	Calls/s/core (max.)	Size (ø) (bytes)	Load/core (ø) (KB/s)	Load/core (max.) (KB/s)
Alegra	2,744	32.18	81	2.54	8.04
Alegra	5,830	12.28	81	0.97	3.07
CTH	3,300	88.87	80	6.94	22.21
S3D MPI I/O	6,400	11.05	85	0.91	2.76
S3D pNetCDF	1,024	11.08	80	0.86	2.77
S3D Fortran	6,400	11.24	94	1.03	2.81

operations during the input and output phase while the others are dominated by their periodical I/O. The observation has lead to several optimizations such as the concept of burst buffers, which absorb the write intense load quickly and drain it during the compute phase of applications [3]. With these estimations about the size and frequency of the tracing events, we can derive the network bandwidth required to transmit the tracing events as shown in Table 8.2. Note that this estimation

corresponds only to the I/O events at the POSIX layer and the SIOX system will certainly require more bandwidth than that when logging is turned on for more layers than just POSIX.

8.5 Data Compression Methods

The basic principle behind all data compression methods is the substitution of repeated symbols or groups thereof by shorter ones that act as synonyms. This is done in a way that the most frequently found symbols get replaced by the shortest synonyms available. Because of this, conventional data compression methods consist of at least two components: a model responsible for calculating the probability distribution of the input symbols, and an encoder which uses this probability distribution to optimally assign the synonyms. While the optimal encoding problem has already been solved (e.g. using Huffman Coding, Arithmetic Coding, etc.), determining an optimal model for generic input data is proven to be unsolvable [4]. Consequently, advanced input models could improve compression rate and thus there is a need for specialized data compression methods depending on the data domain.

8.5.1 Specialized Compression Methods for Trace Data

In contrast to general purpose data compression methods like the well known Lempel-Ziv family [10] or those based on Prediction by Partial Matching (PPM) [1] that accommodate generic input data by sacrificing compression ratio, there exist compression algorithms that specialize in the compression of data for an specific domain. Such algorithms take advantage of a priori knowledge about the structure of the input data to achieve much better compression ratios or performance than those obtained with the former compression methods. In the next sections we will examine the use of two specialized methods for the compression of distributed event-based traces, namely ScalaTrace and C3G, as well as their applicability in the SIOX project.

8.5.2 ScalaTrace

ScalaTrace [6] is a library written in C/C++ that can be used to create, transmit, compress and replay MPI traces. It was created together by the Lawrence Livermore National Laboratory (LLNL) and the North Carolina State University (NCSU). In this paper we will only focus on the compression capabilities of the library. ScalaTrace compresses event-based traces located on distributed nodes and outputs

a single unified trace file. This happens in two steps: the first one is called *intranode* compression and happens every time an event is appended to the local trace of a computing node. The second one is referred to as *internode* compression and it merges and compresses all the traces produced by the previous step. This latter procedure is launched after program termination.

The compressed traces produced by ScalaTrace consist of a list of Regular Section Descriptors (RSDs), with each RSD having the following form:

$$< \text{repetitions}, \text{event}_1, \ldots, \text{event}_n >$$

The repetitions field is the number of times the events 1 to n are repeated, and the events themselves represent either the MPI functions that were called or another RSDs (e.g. to describe nested loops).

8.5.2.1 Intranode Compression

ScalaTrace's algorithm for intranode compression uses a backward search to look for repeated patterns every time a new event is appended to the trace, as illustrated in Fig. 8.5. Since this approach would have a prohibitive time complexity of $O(n^2)$, the range of the search is limited to a window of a user-defined size m. This reduces the complexity of ScalaTrace's intranode compression to a manageable $O(n \cdot m)$, where the window size can be optimized depending on the scenario.

In best-case scenarios, the intranode compression is able to create a mostly constant-length representation of a repeating function call pattern independently of the number of times the pattern occurs. However, if the timing information of each function call is to be kept (as required by projects like SIOX), it will augment the RSD in an incompressible manner as shown in Fig. 8.6. This would lead to a linear growth in the size of the RSD whose magnitude is determined by the repetition number $\times$ the number of events in the RSD.

8.5.2.2 Internode Compression

The algorithm for internode compression unifies the separate traces located on different computing nodes into one single trace. ScalaTrace launches the internode compression as soon as the involved application terminates (call to `MPI_Finalize()`). When this happens, half of the nodes receive the local traces of the other half, merge them with their own local traces, subdivide into two groups and repeat the process until only one node is left with the complete trace. This binary unification process is shown in Fig. 8.7 and detailed in [6]. The merge of two traces has a worst-case time complexity of $O(n^2)$ assuming each of them has n elements.

As with the intranode compression, the use of this approach in projects like SIOX, demands the RSDs to be enriched with spatial information about the events

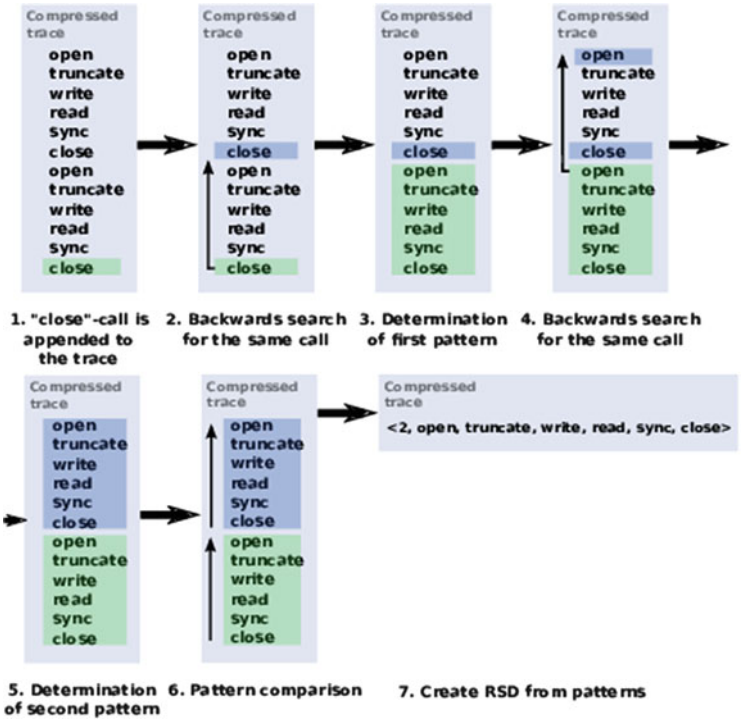


Fig. 8.5 Illustration of ScalaTrace's intranode compression algorithm for $m = 12$

Fig. 8.6 Intranote compression with added timing information

(e.g. node-id, process number, etc.), which again reduces the compressibility of the RSD. ScalaTrace supports the separate compression of the spatial information to help mitigate this effect.

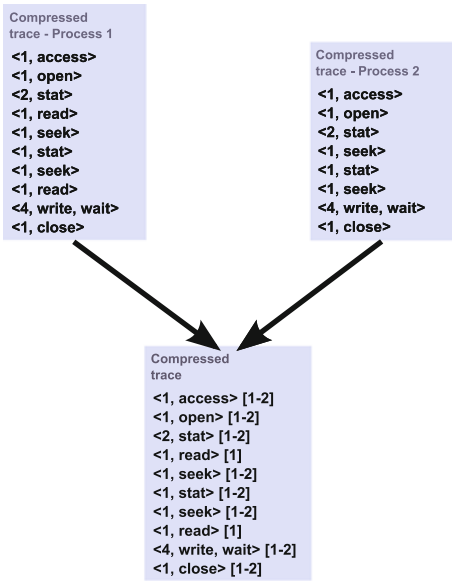


Fig. 8.7 Illustration of ScalaTrace's internode compression

8.5.3 The C3G Library

C3G is a small C++ library that allows the creation and manipulation of compressed complete call graphs. The idea originated in [2] and the library itself is being developed at the Center for Information Services and High Performance Computing (ZIH) of the TU Dresden with the aim of replacing the central data structure of the Vampir software.[1] Current versions of the library support input traces in OTF(2) format. However, given its simplicity, the library is easily extended to support any input format desired.

8.5.3.1 Compressed Complete Call Graphs

A Complete Call Graph (CCG), as defined in [2], is a type of call graph whose nodes describe all function calls, their hierarchy as well as the parameters used when issuing the call. The CCG of a single process takes the form of a partially balanced call tree, where the balancing is achieved by limiting its branching factor. Parallel applications possess one call tree per participating process or thread. The CCG corresponding to the trace shown in Fig. 8.5 is illustrated in Fig. 8.8a. This call graph has two prominent features:

[1] http://www.vampir.eu

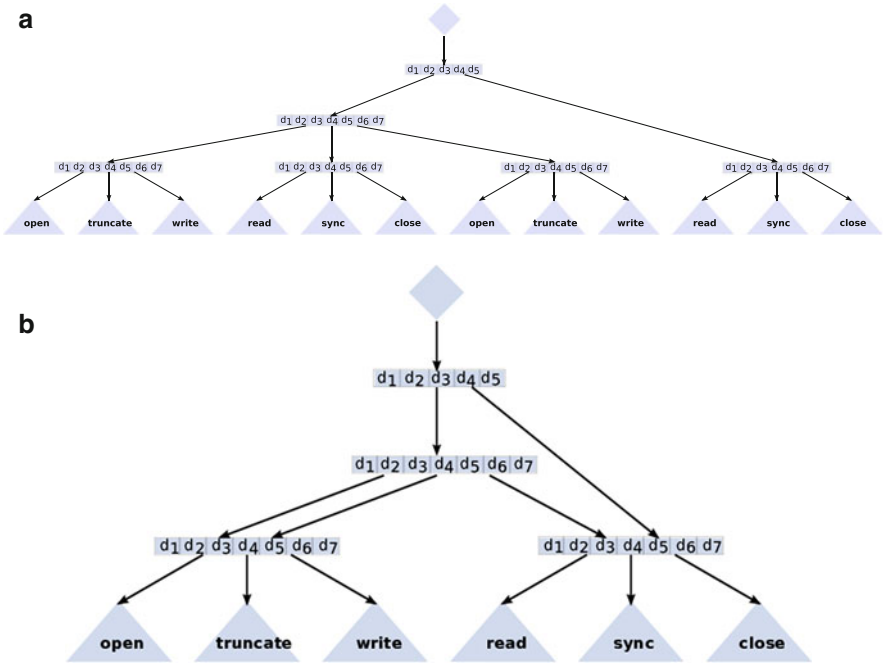


Fig. 8.8 Complete call graph for one process and branching factor 3. (**a**) Uncompressed. (**b**) Compressed

1. The function calls are represented by the tree's leaves.
2. There is a significant number of nodes between the root and the leaves.

In the case of this particular example, function calls are atomic I/O operations that don't trigger any suboperation and that is why they are shown as leave nodes. The additional nodes between the root node and the leaves are inserted on demand to enforce the branching factor and play an important role when optimizing searches. An node with n children contains the following $2n + 1$ timing fields ($d_1 \ldots d_{2n+1}$):

1 start time of the first child call
n durations of the children's calls
$n - 1$ delays between the end and begin of subsequent child calls
1 completion time of the last child call

The compression of a CCG consists of the substitution of repeated nodes (or trees) by pointers to a single instance of the node (or tree). This can be accomplished in $O(\log n)$ using conventional tree search algorithms. Figure 8.8b shows the Compressed Complete Call Graph (cCCG) for the previous example.

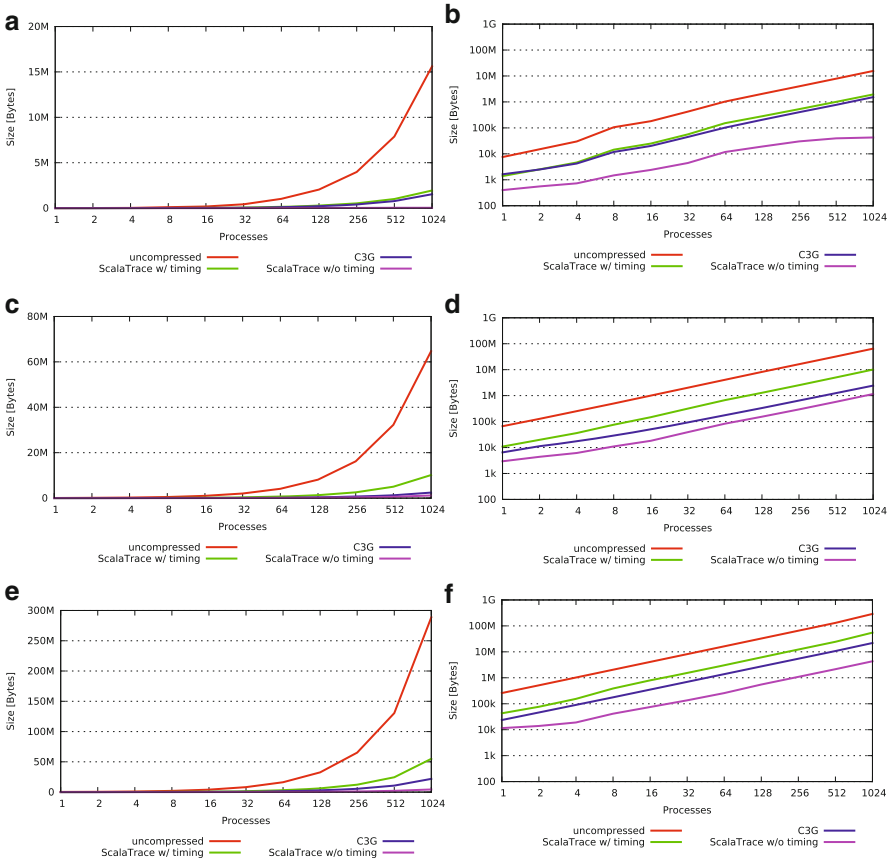


Fig. 8.9 Compression of three different traces using ScalaTrace and C3G. (**a**) S3D pNetCDF. (**b**) S3D pNetCDF (log.). (**c**) Alegra. (**d**) Alegra (log.). (**e**) CTH. (**f**) CTH (log.)

8.6 Evaluating the Compression of ScalaTrace and C3G

The compression of the two libraries was compared using the I/O traces presented in Sect. 8.4. We were interested to see how the size of the final traces evolves as we increase the number of computing nodes producing trace data. This represents one of the main scalability challenges SIOX is facing as explained in Sect. 8.3. For simplicity, we omitted the function parameters when reading the trace data. The window size of ScalaTrace was set to 500 and C3G's branching factor to 30 since bigger values did not improve the compression ratios of these traces.

The results of this experiment are shown in Fig. 8.9a–f, where the x axis is given in logarithmic scale and represents the number of nodes running the application. The red line in the graphics represents the size of the uncompressed trace file used containing the trace information in ASCII format. The green and magenta lines show

the sizes of the output traces compressed using the ScalaTrace algorithm with and without timing information respectively. Finally, the size of the traces compressed with the C3G library is presented in blue. Since the C3G library does not have output capability so far, the sizes presented here refer to the amount of memory occupied by the cCCG structure residing in main memory.

From the results we can clearly see that the best compression is achieved using ScalaTrace without timing information. In this case the trace size stayed almost constant as the number of clients increased and the small growth it experienced was mainly due to the increased spatial information added with each increment in the number of clients. Once we added the incompressible timing information, the optimality of ScalaTrace's compression was compromised and in most cases C3G performed better than ScalaTrace. Using ScalaTrace with timing, the achieved compression ratios[2] varied between 5.5 and 7.0. The C3G library showed a bigger variation range going from 7 for the S3D pNetCDF trace up to 24 for the Alegra trace. The difference in ratios between ScalaTrace and C3G can be explained by the lack of a search window limiting C3G's searches as well as the advantage when comparing the binary space requirements of the cCCG against the ASCII output of ScalaTrace. Figs. 8.9b, d, f present the data using double logarithmic axes to better appreciate the differences between the compression approaches.

8.7 Conclusion and Outlook

In this paper we briefly introduced the problematic of system-wide I/O tracing on HPC systems, characterized the kind of data that has to be managed in such scenario, and examined two specialized compression libraries that might help the SIOX project tackle the problem. Both ScalaTrace and C3G offered promising results that could greatly improve the scalability of SIOX by reducing the size of its traces by an order of magnitude without compromising information content. While ScalaTrace is mainly designed to create a compact trace log that can be used later on to replay the execution of an application, the focus of C3G is to create a compact memory structure to analyze the data. Insofar, the use of one library does not necessarily exclude the other. Further work could be aimed at the integration of the best aspects of both libraries to create a highly scalable framework to manage distributed event-based traces. In our particular case of SIOX, however, the use of either approach would require among other things the modification of the post-mortem components to make them work online. Additionally, the performance overhead incurred by the compression should be tuned to stay within tolerable limits, both regarding the CPU cycles, as well as the transmission delay. It remains

[2]The compression ratio is defined as the uncompressed size divided by the compressed size of the trace [2].

to be seen to which extent the use of trace compression can be achieved within the time frame of the SIOX project.

Acknowledgements We would like to express our gratitude to the German Aerospace Center (DLR) as the responsible agency for the SIOX project as well as to the German Federal Ministry of Education and Research (BMBF) for the financial support under grant `01 IH 11008 A-C`. Our gratitude also extends to Andreas Knüpfer and Joachim Protze from the Center for Information Services and High Performance Computing (ZIH) at the TU Dresden for the expertise provided when we used the C3G library.

References

1. Cleary, J.G., Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Trans. Commun. **32**(4), 396–402 (1984)
2. Knüpfer, A., Nagel, W.E.: Compressible memory data structures for event-based trace analysis. Future Gener. Comput. Syst. **22**, 359–368 (2006). http://dx.doi.org/10.1016/j.future.2004.11.021
3. Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., Maltzahn, C.: On the role of burst buffers in leadership-class storage systems. In: 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), Pacific Grove, pp. 1–11. IEEE (2012)
4. Mahoney, M.: Data compression explained. Available at http://mattmahoney.net/dc/dce.html (2011)
5. Nakka, N., Choudhary, A., Liao, W.K., Ward, L., Klundt, R., Weston, M.I.: Detailed analysis of I/O traces for large scale applications. In: 2012 IEEE 28th Symposium on High Performance Computing (HiPC), Kochi, pp. 419–427 (2009)
6. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: ScalaTrace: scalable compression and replay of communication traces for high-performance computing. J. Parallel Distrib. Comput. **69**, 696–710 (2009)
7. Pieterse, V., Black, P.E.: Algorithms and Theory of Computation Handbook. Available at: http://www.nist.gov/dads/HTML/singleprogrm.html (Dec 2004); Dictionary of Algorithms and Data Structures
8. Sandia National Laboratories: Scalable I/O project I/O traces. Available at: http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data (2009)
9. Wiedemann, M.C., Kunkel, J.M., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W.E., Kluge, M., Mickler, H.: Towards I/O analysis of HPC systems and a generic architecture to collect access patterns. Comput. Sci. Res. Dev. **1**, 1–11 (2012). http://dx.doi.org/10.1007/s00450-012-0221-5
10. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)

Chapter 9
PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis

Zakaria Bendifallah, William Jalby, José Noudohouenou, Emmanuel Oseret, Vincent Palomares, and Andres Charif Rubial

Abstract Identifying performance bottlenecks in applications is crucial to improve their efficiency, but it might be difficult to precisely assess their impact on performance: in particular, two performance problems can interact making it difficult to isolate and therefore to correct them. We propose PAMDA, a methodology to single out performance problems through hierarchical bottlenecks detection. Important potential performance issues are classified in a 'Performance Breakdown Tree' which is used to drive our iterative analysis cycle, prioritizing the most relevant problems. Our system relies on MAQAO toolset and code's differential analysis. While MAQAO is a performance analysis and optimization tool suite, the differential analysis approach, which is implemented through DECAN tool, consists in quantifying performance changes when applying controlled transformations to the target code. Our focus will be on performance issues raised by processors and memory sub-systems in multicore architectures. We will demonstrate the approach on loops extracted from real life HPC applications.

9.1 Introduction

The recent progress of high performance architectures generate new challenges for performance evaluation tools: more complex processors (larger vectors, manycores), more complex memory systems (multiple memory levels including NUMA, multiple level prefetch mechanisms), more complex systems (large increase in core counts up to several hundred of thousands now) are all key issues which need to be simultaneously optimized to get a decent performance level.

To work properly, all of these mechanisms require specific properties from the target code. For example, good exploitation of memory hierarchies relies on good spatial and temporal locality within the target code. The lack of such

Z. Bendifallah • W. Jalby (✉) • J. Noudohouenou • E. Oseret • V. Palomares • A.C. Rubial
Exascale Computing Research, Versailles, France

University of Versailles Saint-Quentin-en-Yvelines, Versailles, France
e-mail: zakaria.bendifallah@exascale-computing.eu; William.Jalby@uvsq.fr

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2013*,
DOI 10.1007/978-3-319-08144-1_9

properties induces variable performance penalties: such combinations (mismatch between hardware and software) are denoted performance pathologies. Most of them have been identified (cf. Table 9.1) and efficient workarounds are well known. The current generation of performance tools (TAU [21], PerfExpert [7], VTune [12], Acumem [1], Scalasca [10], Vampir [19]) is excellent at detecting such pathologies although some are fairly specialized: for instance, Scalasca/Vampir mainly addresses MPI/OpenMP issues, requiring the combined use of several tools

Table 9.1 A few typical performance pathologies

Pathologies	Issues	Workarounds
ADD/MUL balance	ADD/MUL parallel execution (of fused multiply add unit) underused	Loop fusion, code rewriting e.g. Use distributivity
Non pipelined execution units	Presence of non pipelined instructions: div and sqrt	Loop hoisting, rewriting code to use other instructions e.g. x86: div and sqrt
Vectorization	Unvectorized loop	Use another compiler, check option driving vectorization, use pragmas to help compiler, manual source rewriting
Complex control flow graph in innermost loops	Prevents vectorization	Loop hoisting or code specialization
Unaligned memory access	Presence of vector-unaligned load/store instructions	Data padding, use pragma and/or attributes to force the compiler
Bad spatial locality and/or non stride 1	Loss of bandwidth and cache space	Rearrange data structures or loop interchange
Bad temporal locality	Loss of perf. due to avoidable capacity misses	Loop blocking or data restructuring
4K aliasing	Unneeded serialization of memory accesses	Adding offset during allocation, data padding
Associativity conflict	Loss of performance due to avoidable conflict misses	Loop distribution, rearrange data structures
False sharing	Loss of bandwidth due to coherence traffic and higher latency access	Data padding or rearrange data structures
Cache leaking	Loss of bandwidth and cache space due to poor physical-virtual mapping	Use bigger pages, blocking
Load unbalance	Loss of parallel perf. due to waiting nodes	Balance work among threads or remove unnecessary lock
Bad affinity	Loss of parallel perf. due to conflict for shared resources	Use numactl to pin threads on physical CPUs
High number of memory streams	Too many streams for hardware prefetcher or conflict miss issues	See conflict misses
Lack of loop unrolling	Significant loop overhead, lack of instruction-level parallelism	Try different unrolling factors, unroll and jam for loops nest, try classical affinities (compact, scatter, etc.)

to get a global overview of all of the performance pathologies present in an application.

Most of the current tools do not provide any direct insight on the potential cost of a pathology. Furthermore, the user has no idea about what the potential benefit of optimizing his code to fix a given pathology is. These two points prevent him from focusing on the right issue. For example, let us consider a program containing two hot routines A and B, respectively consuming 40 and 20 % of the total execution time. Let us further assume that the potential achievable performance gain on A is 10 % while on B it is up to 60 %. The overall performance impact on B is up to 60 % * 20 % = 12 % while on routine A, it is at best 10 % * 40 % = 4 %. As a consequence, it is preferable to focus on routine B. Additionally, the user has no clue of what the current performance level is, compared with the best one achievable, i.e. he may not know when optimizing is worth the investment.

In general, the situation is even worse since a simple loop may simultaneously exhibit several performance pathologies. In such cases, most of the tools cited above give no hint to the user of which ones are dominant and really worth fixing. For instance, a loop can suffer from both a high miss rate and the presence of costly Floating-Point (FP) operations such as div/sqrt: trying to improve the hit rate does not improve the performance if the dominant bottleneck is the div/sqrt operations.

In this paper, we present a coherent set of tools (MicroTools [6], CQA [4, 14, 17], DECAN [13], MTL [8]) to address this lack of user's guidance in the tedious and difficult task of program optimization. These tools are integrated in a unified environment (PAMDA) to help the user to quickly identify performance pathologies and to assess their cost and impact on the global performance. Depending upon which performance pathologies is to be fixed, different techniques (static analysis, value profiling, dynamic analysis) appear to be more appropriate and give a more accurate answer: for example, detecting a badly strided access is immediate through value tracing of array addresses while the same task is extremely tedious when only using static analysis or hardware counters. Anyway, such array access tracing should only be triggered when necessary due to its high cost. In this paper, we focus on providing performance insight at the core level and parallel OpenMP structures. Our analysis can be combined with MPI analysis provided by tools such as Scalasca, TAU or Vampir.

Through the integrated environment PAMDA, we aim at providing the following contributions:

- Getting a global hierarchical view of performance pathologies/bottlenecks.
- Getting an estimate of the impact of a given performance pathology taking into account all other present pathologies.
- Demonstrating that different specialized tools can be used for pathology detection and analysis.
- Performing a hierarchical exploration of bottlenecks according to their cost: the more precise but expensive tools are only used on specific well chosen cases.

Section 9.2 presents a motivating example in detail. Section 9.3 details the various key components of PAMDA while Sect. 9.4 describes the combined use of these different tools. Section 9.5 describes some experimental use of PAMDA. Section 9.6 gives an overview of related works and the added value of the PAMDA system. Finally, Sect. 9.7 gives conclusions and future directions for improvement.

9.2 Motivating Example

Figure 9.1 presents the source code of one of the hottest loops extracted from POLARIS(MD) [20]: a molecular dynamics application developed at CEA DSV. POLARIS(MD) is a multiscale code based on Newton equations: it has been successfully used to model Factor Xa involved in thrombosis.

This loop simultaneously presents a few interesting potential pathologies:

- Variable loop trip count.
- Fairly complex loop body which might lead to inefficient code generation by the compiler.
- Presence of div/sqrt operations.
- Strided and indirect access to arrays (scatter/gather type).
- Multiple simultaneous reduction operations leading to inter iteration dependencies.

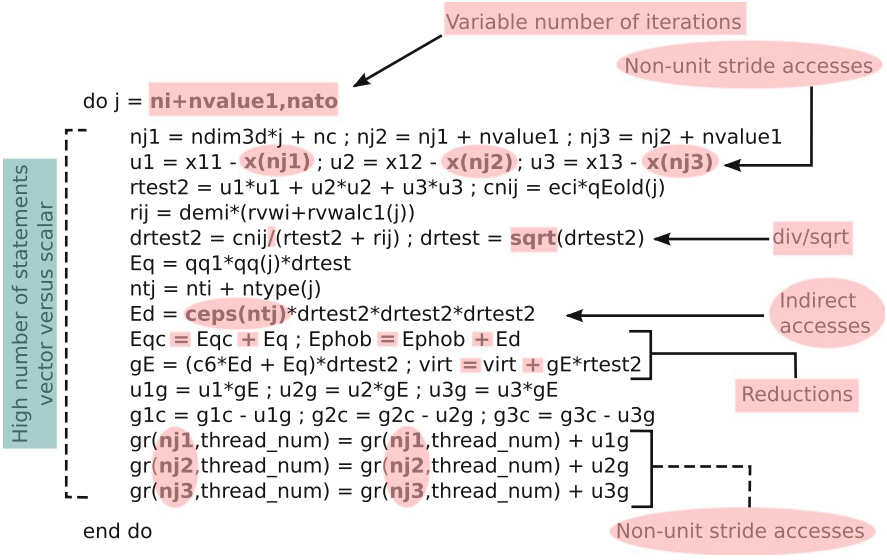


Fig. 9.1 A Fortran source code sample and its main performance pathologies highlighted in *pink*

All these pathologies can be directly identified by simple analysis of the source code. The major difficulty is to assess the cost of each of them and therefore to decide which should be worked on.

A first value profiling of the loop iteration count reveals that the trip count is widely varying between 1 and 2,000. However the amount of time spent in the small (less than 150 iterations) loop trip count instances remains limited to less than 10 %. The remaining interval of loop trip counts is further divided into 10 deciles and one representative instance is selected for each of them. Further timings on analyzing loop trip count impact indicate that the average cost per iteration globally remains constant independently from the trip count. Therefore, the data size variation seems to have no impact on performance: the same optimization techniques should apply for instances having a loop trip count between 150 and 2,000.

The static analyzer (see Fig. 9.3) provides us with the following key information: in the original version, neither Load/Store (LS) operations nor FP ones are vectorized. It further indicates that due to the presence of div/sqrt operations, the FP operations are the main bottlenecks. It also points out that even if the FP operations were vectorized, the bottlenecks due to div/sqrt operations would remain. However this information has to be taken with caution since the static analyzer assumes that all data accesses are ideal, i.e. performed from L1.

Dynamic analysis using code variants generated by DECAN is presented in Fig. 9.2. Initially, the original code (in dark blue bars) shows that FP operations (see FP versus LS DECAN variants) clearly are the dominating bottlenecks. Furthermore, the good match between CQA and REF clearly indicates that analysis

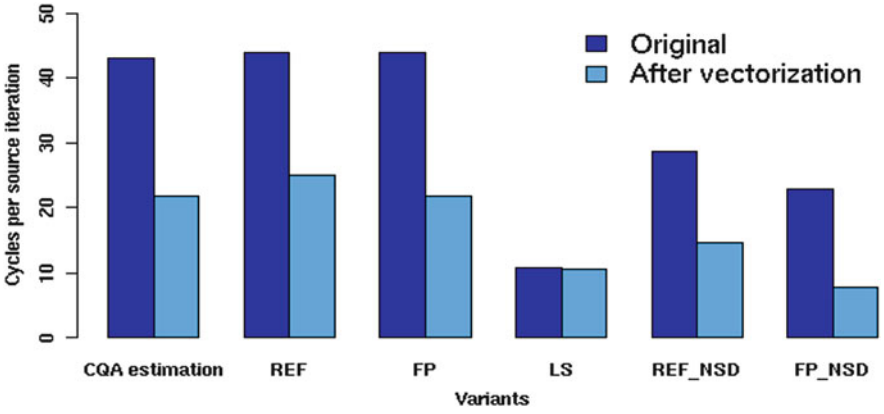


Fig. 9.2 Comparing static estimates obtained by CQA with dynamic measurements performed on different code variants generated by DECAN of both of the original and vectorized versions: **REF** is the reference binary loop (no binary modifications introduced by DECAN), **FP** (resp. **LS**) is the DECAN binary loop variant in which all of the Load/Store (resp. FP) instructions have been suppressed, **REF_NSD** (resp. **FP_NSD**) is the DECAN binary loop variant in which only FP div and sqrt instructions (resp. all of the Load/Store and FP div and sqrt instructions) have been suppressed. The y-axis represents the number of cycles per source iteration: lower is better

made by CQA is valid and pertinent. Optimizing this loop is simply obtained by inserting the SIMD pragma '`!DEC$ SIMD`', which forces the compiler to vectorize FP operations. However, the compiler does not vectorize loads and stores due to the presence of strides and indirect access. Rerunning DECAN variants of this optimized version (see light blue bars in Fig. 9.2) shows that even for this optimized version FP operations still remain the key bottleneck (comparison between LS and FP). Therefore, there is no point in optimizing data access, the only hope of optimization lies in improving div/sqrt operations: for example SP instead of DP. Unfortunately, such a change would alter the numerical stability of the code and cannot be used.

The major lesson to be drawn from this case study is that a combined use of CQA and DECAN allows us to quickly identify the optimization to be performed and also gives us a clear halt on tackling other pathologies without impacting overall performance.

9.3 Ingredients: Main Tool Set Components

Performance assessment issues require robust methodologies and tools. Therefore, in order to systematically provide programmers with a performance pathology hierarchy and its related costs, the current work considers two toolsets: MicroTools, for microbenchmarking the architecture, and the MAQAO [4, 14, 17] framework, which is a performance analysis and optimization tool suite.

MAQAO's goal is to analyze binary codes and to provide application developers with reports to optimize their code. The tool mixes both static (code quality evaluation) and dynamic (profiling, characterization) analyses based on the ability to reconstruct low level (basic blocks, instructions, etc.) and high level structures such as functions and loops. Another MAQAO key feature is its extensibility. Users easily write plugins thanks to an embedded scripting language (Lua), which allows fast prototyping of new MAQAO-tools.

From MAQAO, PAMDA extensively uses three tools including the Code Quality Analyzer tool (CQA) exposed in Sect. 9.3.2, the Differential Analysis framework (DECAN) presented in Sect. 9.3.3, and finally the Memory Tracing Library (MTL) in Sect. 9.3.4. We briefly present the main contribution of each of these tools to PAMDA and then describe their major characteristics.

9.3.1 MicroTools: Microbenchmarking the Architecture

Microbenchmarking [3, 16, 23] is an essential tool to investigate the real potential of a given architecture: more precisely, in PAMDA, microbenchmarking is first used to determine both FP units performance and achievable peak bandwidth of

various hardware components such as cache/RAM levels, and second to estimate the potential cost of various pathologies (unaligned access, 4K aliasing, high miss rate, etc.).

For achieving these goals, PAMDA relies on `MicroTools`, consisting of two main components: `MicroCreator` tool automatically generates a set of benchmark programs, while `MicroLauncher` framework executes them in a stable and closed environment.

9.3.2 CQA: Code Quality Analyzer

In PAMDA, the CQA framework is used first for providing a performance target under ideal data access conditions (all operands are supposed to be in L1), second for providing a bottleneck hierarchy analysis between the various hardware components of the core (FP units, load/store ports, etc.) and third for detecting some performance pathologies (presence of inter iterations dependencies, div/sqrt operations) which are worth investigating via specialised DECAN variants. The ideal assumption (all operands in L1) is essential for CPU bound codes such as the POLARIS(MD) loop studied in the previous section. For memory bound loops, it needs to be complemented with a dynamic analysis.

CQA is a static analysis tool directly dealing with binary code. It extracts key characteristics, and detects potential inefficiencies. It provides users with general code metrics such as details on basic loop characteristics, the number of instructions, μ`ops`, and used XMM/YMM vector registers. CQA also allows users to obtain more in-depth information on the loop execution on the target architecture. For example, the tool provides a reliable front-end pipeline execution report, which is an estimated number of cycles spent during each front-end pipeline stage. The tool gives the same type of report for the back-end. Finally, CQA provides a cycle estimate of loop body performance under ideal conditions: all operands in L1, no branches and infinite loop count (steady state behavior).

CQA is able to report both low and high level metrics/reports (Fig. 9.3). For example, when a loop is not fully vectorized, the high level report provides a potential speedup (if all instructions were vectorized) and corresponding hints (compiler flags and source transformations). For the same loop, some low level metrics/reports show the breakdown of vectorization ratios per instruction type (loads, stores, ADDs, etc.) giving the user a more in-depth view of the issue.

CQA supports Intel 64 micro-architectures from Core 2 to Ivy Bridge.

9.3.3 DECAN: Differential Analysis

In PAMDA, DECAN is used for quantitatively assessing performance pathologies impact. The general idea is fairly simple: a given pathology is associated with

```
Unroll factor: 1 or NA

*************************************
             Back-end
*************************************
       P0     P1    P2    P3    P4    P5
FU     FP ×/÷ FP +  LD1   LD2   ST    OTH.
Uops   18.00  17.00 9.50  9.50  3.00  6.00
Cycles 43.00  17.00 9.50  9.50  3.00  6.00

Cycles executing div or sqrt
instructions: 20-43 (second value used
for L1 performances)
Longest recurrence chain latency
(RecMII): 3.00

*************************************
       Vectorization ratios
*************************************
All      : 0%
Load     : 0%
Store    : 0%
Mul      : 0%
add_sub  : 0%
Other    : 0%

*************************************
      Vector efficiency ratios
*************************************
All      : 25%
Load     : 25%
Store    : 25%
Mul      : 25%
add_sub  : 25%
Other    : 25%

*************************************
        If all data in L1
*************************************
cycles: 43.00
FP operations per cycle: 0.81 (GFLOPS
at 1 GHz)
(…)
Cycles if fully vectorized: 21.50
```

Fig. 9.3 CQA output

the presence of a given subset of instructions, for example div/sqrt operations, then DECAN generate a binary version of the loop in which the corresponding instructions are deleted or properly modified. This altered binary is measured and compared with the original unmodified version.

The resulting binary does not in general preserve semantics, i.e. numerical values generated with DECAN variants are not identical to the original ones. For our performance analysis objective, this is not a critical issue but for the subsequent program execution, control behavior might be altered. To avoid such problems, the original loop is systematically replayed after the execution of the modified binary in order to restore correct memory values.

DECAN starts by using static analysis on the target loop produced by CQA. The goal is to select instruction subsets to be transformed, as the selection process is driven by the desired type of behavior to highlight. Afterward, instructions are carefully transformed in a manner that minimizes unwanted side effects that may disturb the observations, such as changes in the code layout and instruction dependencies. It also inserts some monitoring probes to be able to accurately compare the modified part of the code with the original one. Also, and as stated earlier, DECAN is built on top of the MAQAO framework, hence, it uses the MAQAO disassembler/patcher to forward modifications on the instructions.

Using DECAN's features, PAMDA generates altered binaries, thereby splitting performance problems between CPU, memory, and OpenMP issues. Table 9.2 presents a range of loop variants used within the methodology discussed in Sect. 9.4.

Table 9.2 DECAN variants and transformations

Variant	Type of SSE/AVX instructions involved	Transformation
LS	All arithmetic instructions	Instruction deleted
FP	All memory instructions	Instruction deleted
DL1	All memory instructions	Instruction operands modified to target a unique address
NODIV	All division instructions	Instruction deleted
NORED	All reduction instructions	Instruction deleted
S2L	All store instructions	Converted into load instructions
NO_STORE	All store instructions	Instructions are deleted

9.3.4 MTL: Memory Tracing Library

Within PAMDA, MTL provides specific analysis of pathologies related to data access patterns in particular stride values, alignment characteristics, data sharing issues in multi-threaded codes, etc. MTL works by tracing addresses and by generating compact representations of data access patterns. MTL is not limited to innermost loops but directly deals with multiple nested loops, allowing to detect more subtle pathologies: for example, row major instead of column major accesses for a Fortran array (stored column wise) are automatically detected. To perform these analysis, MTL uses the MAQAO Instrumentation Language (MIL) [9]. This language makes the development of program analysis tools based on static binary instrumentation easier. In fact, MIL is a specific language for object-oriented and event-directed domains to perform binary instrumentation at a high level of abstraction using structural objects (functions, loops, etc.), events, filters, and probes.

9.4 Recipe: PAMDA Tool Chain

Individual tools are the building blocks that PAMDA glue together through a set of scripts (cf. all the diagrams). These scripts are under development but most of the principles have been already evaluated. Figure 9.4 presents PAMDA overall organization, which includes application profiling, cost analysis, structural checks, CPU and memory subsystems evaluation, and finally OpenMP evaluation for parallel applications. The current section describes PAMDA's components.

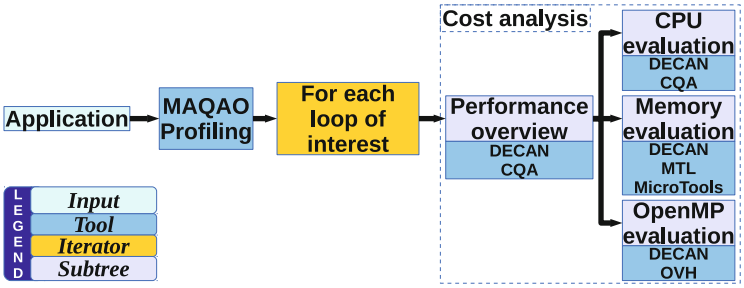


Fig. 9.4 PAMDA overview

9.4.1 Hotspot Identification

To limit the processing cost, we focus on the most time consuming portions of the code. Our target loops are defined as the loops with a cumulated execution time exceeding 80 % of the total execution time. It should be noted that with such an aggregated measurement, we can end up with a large number of loops with small individual contributions. Such target loops are identified using `MAQAO sampling`.

9.4.2 Performance Overview

The PAMDA approach divides performance bottlenecks into two main categories (Fig. 9.5): memory subsystem and CPU. Then, their respective contribution to the overall execution time is quantified using DECAN transformations `LS` (assessing memory subsystem performance) and `DL1` (assessing CPU subsystem performance). The ratio of these contributions reveals whether the loop is memory or/and CPU bound. Ideally, pipeline and out of order mechanisms insure that cycles spent for memory accesses and for arithmetic operations perfectly overlap: as a result, the time taken by `REF` should be the maximum time taken either by `LS` or `DL1`. In such a case, only the slower component needs optimizing. If the time taken by `LS` and `DL1` is similar, the workload is said to be balanced: optimizing both components is necessary to improve the loop's performance. Finally, when cycles taken by the memory and CPU components are poorly covered by one another (`unsaturation`), optimizing either of them can be sufficient to gain overall performance.

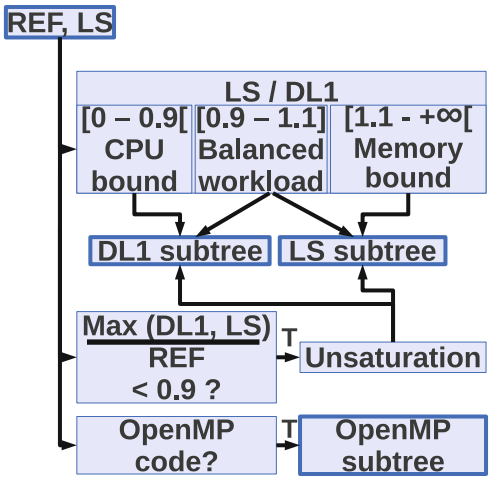


Fig. 9.5 Performance investigation overview. '?' represents a condition and 'T' means the condition is True, otherwise it is False ('F')

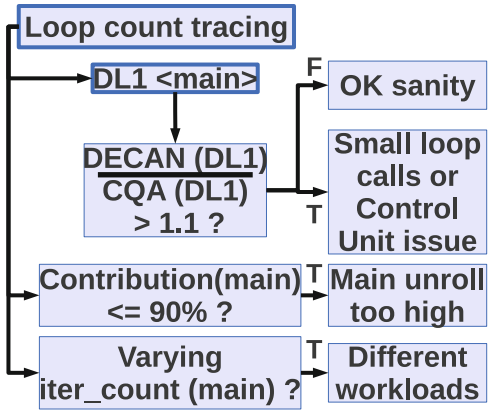


Fig. 9.6 Detecting structural issues. 'Iter_count' illustrates iteration counts from the main loop

9.4.3 Loop Structure Check

Loop structure issues can be detrimental to performance, and may be detected using DECAN's loop trip counting feature. Indeed, in the case of unrolling or vectorization, peel and tail scalar codes may have to be generated to cover for remaining iterations. If too much time is spent in these peel and tail codes, this might indicate the unroll factor is too high with respect to the source loop iteration count. To detect such cases, loop trip counts for each version (peel/tail/main) are determined, and we check whether the main loop is processing at least 90 % of the source code iterations (Fig. 9.6).

In some cases, the number of iterations per loop instance may not be large enough to fully benefit from unrolling or vectorization. This is easily highlighted by comparing the dynamic execution time of the DL1 DECAN variant with the CQA estimate, as the latter assume an infinite trip count. The difficulty to optimize such loops is exacerbated when the loop trip counts are not constant.

9.4.4 CPU Evaluation

Besides data accesses, CPU performance may be limited by other pathologies such as long dependency chains (`deps`), reductions (`RED`), scalar instructions or long latency floating point operations (`div`): these pathologies can be detected through the combined use of CQA and DECAN (Fig. 9.7). The front-end can also slow down the execution by failing to provide the back-end with micro-operations at a sufficient rate. Comparing their contribution to L1 performance (`DL1`) is a cost-effective way to identify such problems. Finally, CQA can provide us with estimations of the effect of vectorizing a loop. We precisely quantify CPU related issues, enabling us to reliably assess potential for optimizations such as getting rid of divisions, suppressing dependencies or vectorizing. This information can guide the user's optimization decisions.

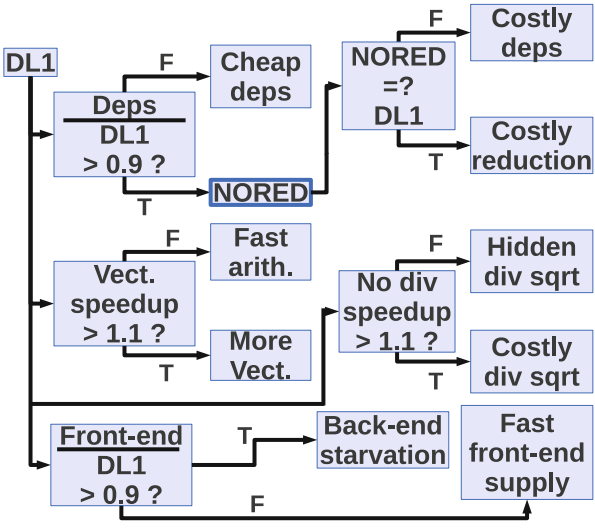


Fig. 9.7 DL1 subtree: CPU performance evaluation. Except DL1 and NORED, others metrics used by conditioners are extracted from CQA static analysis

Table 9.3 Various vector/scalar load bandwidths estimation in bytes per cycle for each memory level (Sandy Bridge E5-2680)

	Instruction	L1	L2	L3	RAM
AVX	vmovaps	31.74	15.05	10.81	5.10
	vmovups	31.73	14.96	10.81	5.10
SSE	movaps	30.72	18.16	10.80	5.14
	movups	29.53	17.07	10.79	5.23
	movsd	15.67	11.55	10.61	5.36
	movss	7.91	6.65	6.39	4.97

9.4.5 Bandwidth Measurement

Data access rates from different cache levels/RAM highly depend on several factors, such as the instructions used or the access pattern.

To this end, we generate microkernels loading data in an ideal stream case, testing different configurations for load operations, with or without various software prefetch instructions, and/or splitting the accessed data in streams accessed in parallel. We also force misaligned addressing for `vmovups` and `movups`. Finally, we use `Microlaunch` to run these benchmarks for each level of the memory hierarchy.

On our target architecture, 128-bit SSE load instructions could roughly achieve the same bandwidth as 256-bit AVX (Table 9.3) throughout the whole memory hierarchy. Except for `movss`, all instructions could attain similar bandwidths in L3 and RAM: only the type of instruction really matters for data accesses from L1 or L2, and data alignment is not as relevant as it once was.

9.4.6 Memory Evaluation

Memory performance can be quite complex to evaluate. We use MTL to find the different access patterns and strides for each memory group (as defined by the grouping analysis [13]). Memory accesses typically are more efficient when targeting contiguous bytes, while discontiguous accesses reduce the spatial locality of data. The worst case scenario is having large and unpredictable strides, as hardware prefetchers may not be able to function properly. MTL also provides the data reuse distance, allowing the temporal locality evaluation of groups (Fig. 9.8).

Once potential performance caveats are identified, we can use DECAN transformation `del-group` to single out offending groups and quantify their contribution to the `LS` variant global time. Comparing the bandwidth measured for each group with the bandwidth obtained in ideal conditions in the bandwidth measurement phase may then provide us with an upper limit on achievable performance.

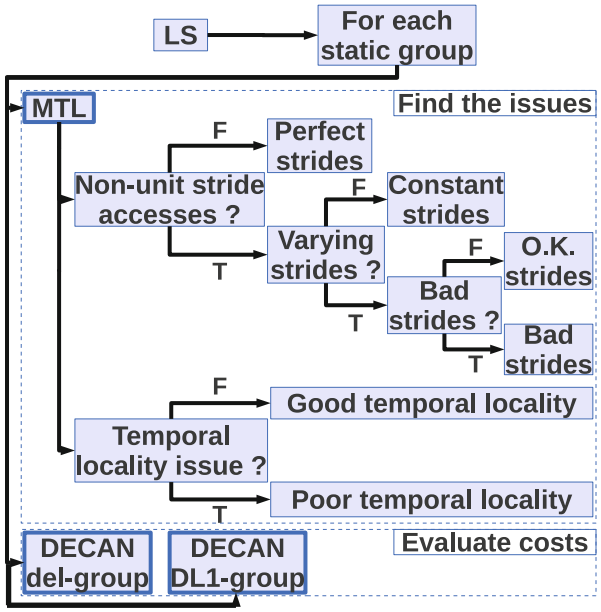


Fig. 9.8 LS subtree: Memory performance evaluation

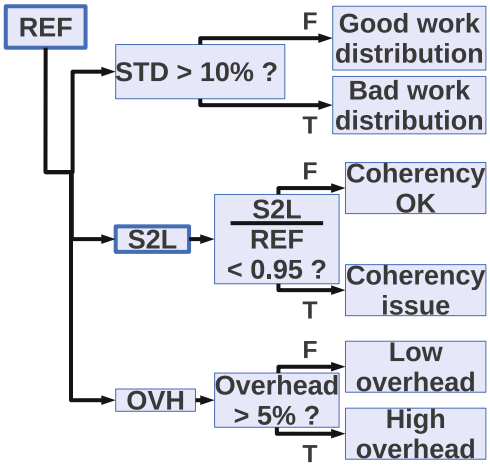


Fig. 9.9 OpenMP performance tree: STD represents the standard deviation between threads while the OVH branch stands for OpenMP Overhead evaluation

9.4.7 OpenMP Evaluation

Some issues are specific to parallel programs using OpenMP (Fig. 9.9). The standard deviation (`STD`) of the execution time for each thread points out workload

imbalances. It is particularly important that no thread takes significantly longer than others to compute its working set, as loop barriers may then highly penalizing stalls. Another issue is excessive cache coherency traffic generated by store operations on shared data. Transformation `S2L` converts all stores to loads: we can quantify coherency penalties by comparing `S2L` with `REF`. Furthermore, the OpenMP Overhead (OVH) module of MAQAO is able to measure the portion of time spent in OpenMP routines, providing an OpenMP overhead metric.

9.5 Experimental Results

We applied our methodology on two scientific applications: PN and RTM. The analysis processes and test results are presented below.

9.5.1 PN

PN is an OpenMP/MPI kernel used at CEA (French Department of Energy). Hot loops are memory bound and are ideal to stress tools dedicated to memory optimizations.

All tests are performed on a two-socket Sandy-Bridge machine, composed of two Intel E5-2680 processors with eight physical cores each.

The profiling done on the initial MPI version of the code presents four loops consuming more than 8 % of the global execution time each. Because of a lack of space, we only study the first one, but the three other loops have a similar behavior.

According to the methodology, the next step consists in gaining more insight on the loop characteristics through *performance overview*, hence, the `LS` and `DL1` DECAN variants are used. The corresponding results shown in Fig. 9.10 indicate a strong domination of data accesses, with the `LS` curve being well over the `DL1` curve and matching the `REF` one. Consequently, the investigation follows the `LS` subtree.

In order to get more information on data accesses, we use MTL. Six instruction groups are detected but only three of them contain relevant SSE instructions dealing with FP arrays. Experimental results in Table 9.4 illustrates MTL output, which uses i1, i2, and i3 to represent loop indices leading to conclude the considered piece of code contains at least a triple-nested loop. Table 9.4 analysis indicate a simple access pattern for group G1 (stride 1) and, for groups G6 and G5, more complicated patterns which need to be optimized. As a result, in this step we are able to characterize our memory accesses with precision. Though, it leaves us with two accesses and no possibility to know which one is the most important. At this point, we return to our notion of ROI provided through Differential analysis and apply the DECAN `del-group` transformation for each of the three selected groups. The

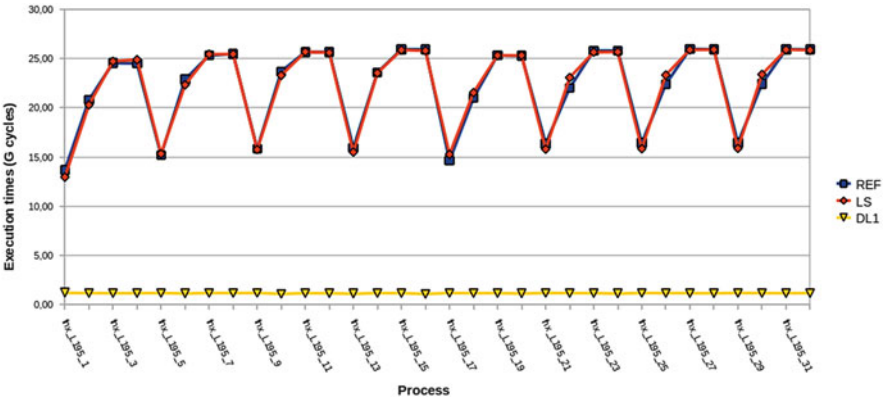


Fig. 9.10 Streams analysis on PN. The REF curve correspond to performance of the original code. The LS (resp. DL1) curve corresponds to the DECAN variant where all FP instructions have been suppressed (resp. all data accesses are forced to come out of L1)

Table 9.4 PN MTL results for the three most relevant instruction groups. i1, i2, and i3 represent loop indices

Group	Instructions	Pattern
G1	Load (Double)	8*i1
G6	Load (Double)	8*i1+217600*i2+1088*i3
G5	Store (Double)	8*i1+218688*i2+1088*i3

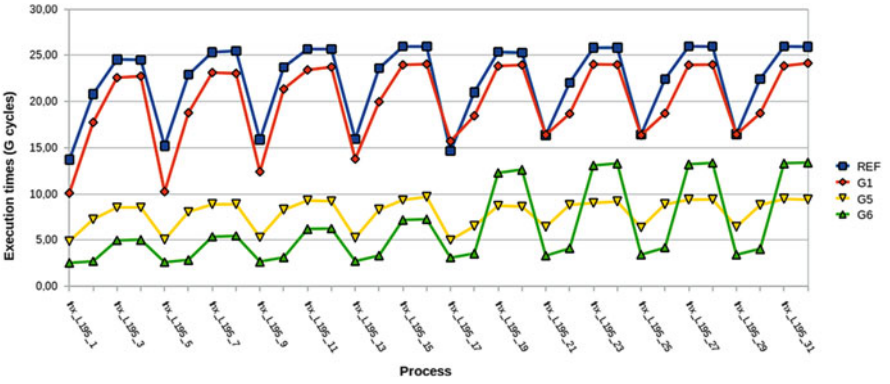


Fig. 9.11 Group cost analysis on PN. Each group curve corresponds to performance of the loop while the target group is deleted. The original code performance (REF) is used as reference

`del-group` results shown in Fig. 9.11 clearly indicate that G6 is the most costly group by far: it should be our first optimization target.

With the finding of the delinquent instruction group, the analysis phase comes to its end. The next logical step is to try and optimize the targeted memory access. Fortunately, the information given by MTL reveals an interesting pathology. The access pattern of the instruction of interest has a big stride in the innermost loop (1088*i3) and a small one in the outermost loop (8*i1). In order to diminish the

access penalty we perform loop interchange between the two loops, which results in a considerable performance gain at the loop level with a speedup of $7.7\times$ and consequently a speedup of $1.4\times$ on the overall performance of the application.

9.5.2 RTM

Reverse Time Migration (RTM) [5] is a standard algorithm used for geophysical prospection. The code used in this study is an industrial implementation of the RTM algorithm by the oil & gas company TOTAL.

Our RTM code operates on a regular 3D grid. More than 90 % of the application execution time is spent in two functions, `Inner` and `Damping`, which execute similar codes on two different parts of the domain: `Inner` is devoted to the core of the domain while `Damping` is used on the skin of the domain. Standard domain decomposition techniques are used to spread the workload on multicore target machines. Since the grid is uniform, load balancing can be easily tuned by using rectangular sub-domain decomposition and by properly adjusting the sub-domain size.

All experiments are done on a single socket machine, which contains a quad-core Intel Xeon E3-1240 processor with a cache hierarchy of 32 KB (L1), 256 KB (L2) and 8 MB (shared L3).

Step 1: The original version of the code is provided with a default non-optimized blocking. The first analysis on the OpenMP subtree reveals an imbalanced work sharing. A second analysis done at the level of the *performance overview* subtree shows that the code is highly bounded by memory operations. In order to fix this, we focus on the blocking strategy. As a result it turns out that the default block size is responsible for both the load imbalance between threads and the bad memory behavior. We can then select a strategy which provides a good balance at the work sharing level as well as a good trade-off between the LS and FP streams. However, we note that, to obtain an optimal strategy, a more dedicated tool should be used.

Step 2: The second step of the analysis consists of going further in the OpenMP subtree and checking how the RTM code performs in term of coherency. As explained earlier, the structure of the code induces a non-negligible coherency traffic. Figure 9.12 shows experimental results after applying the S2L transformation on RTM. While the x-axis details loops respectively identified from `Inner` and `Damping` functions, the y-axis represents speedups over the original loops. The results indicate a negligible gain due to canceling potential coherency modifiers and a minimal gain, observed on two loops, due to a complete deletion of the stores. Consequently, we can conclude that maintaining the overall coherency state remains negligible, therefore, there would be no point in going further in this direction.

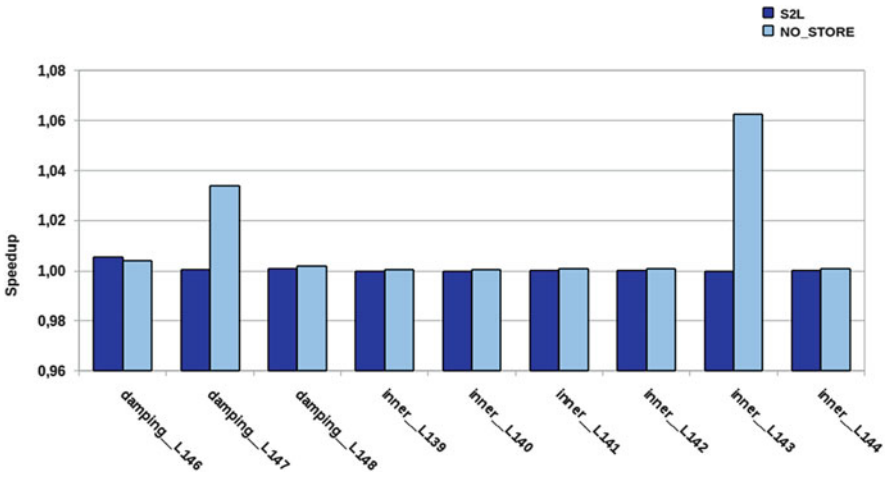


Fig. 9.12 Evaluation of the cost of cache coherence protocol. The S2L variants show similar performance as their corresponding reference versions. The NO_STORE variants show similar performances also, except for two loops which present a relatively non negligible store cost

9.6 Related Work

Improving an application's efficiency requires identifying performance problems through measurement and analysis but assessing bottlenecks impact on performance is much harder. To achieve that, most researchers consider a qualitative approach. TAU [21] represents a parallel performance system that addresses diverse requirements for performance observation and analysis. Although performance evaluation issues require robust methodologies and tools, TAU only offers support to the performance analysis in various ways, including instrumentation, profiling and trace measurements.

Tools such as Intel VTune [12], GNU profiler (Gprof) [11], Oprofile [15], MemSpy [18], VAMPIR [19], and Scalasca [10] provide considerable insight on the application's profile. In term of methodology Scalasca, for instance, proposes an incremental performance-analysis procedure that integrates runtime summaries based on event tracing. While these tools help hardware and software engineers find performance pathologies, significant manual performance tuning remain for software improvements, for example, selecting instructions in particular part of a program.

PerfExpert [7], HPCToolkit [2], and AutoSCOPE [22] pinpoint performance bottlenecks using performance monitoring events. Furthermore, while PerfExpert suggests performance optimizations, AutoSCOPE extends PerfExpert by automatically determining appropriate source-code optimizations and compiler flags. Contrary to PAMDA, the considered tools do not provide a methodology presenting the cost related to the identified bottleneck. ThreadSpotter also helps a programmer

by presenting a list of high level advice without addressing return on investment issues: what to do in case of multiple bottlenecks? How much do bottlenecks cost?

Interestingly in [24,25], the authors present an automated system that fingerprints the pathological patterns of the hardware performance events and identifies the pathologies in applications, allowing programmers to reap the architectural insights. The proposed technique is close to the current work and includes pathology description through microbenchmarks as well as pathology identification using a decision tree. However, in order to evaluate usual performance pathologies, PAMDA additionally integrates pathology cost analysis.

The above survey indicates that performance evaluation requires a robust methodology, but traditional methods do not help much with coping with the overall hardware complexity and with guiding the optimization effort. Also, previous works focus on performance bottleneck identification providing optimization advice without providing potential gains. The previous factors motivate to consider PAMDA as the only methodology combining both qualitative and quantitative approaches to drive the optimization process.

9.7 Conclusion and Future Work

Application performance analysis is a constantly evolving art. The rapid changes in the hardware mixed with new coding paradigms force analysis tools to handle as many pathologies as possible. This can only be achieved at the expense of usability. At the end, application developers work with extremely powerful tools but they have to face significant differences and difficulties to use them.

This paper illustrates the usefulness of performance assessment combining static analysis, value profiling and dynamic analysis. The proposed tool chain, PAMDA, helps the user to quickly identify performance pathologies and assess their cost and impact on the global performance.

The goal in using PAMDA is to make sure that the right effort is spent at each step of the analysis and on the right part of the code. Furthermore, we try to create some synergy between different tools by combining them in a unified environment. We provide some case studies to illustrate the overall analysis and optimization process. Experimental results clearly demonstrate PAMDA's benefits.

Obviously, the provided methodology is far from being finished. Our constant challenge is to keep improving it as well as working towards full automation. We also aim to enlarge it for other kind of paradigms through the integration of analyses provided by complementary tools such as Scalasca, Vampir and TAU. Additionally, refining optimization investigations is crucial in order to make it more user-friendly.

Acknowledgements We would like to thank Michel Masella for the access to his POLARIS(MD) code and Henri Calandra and Asma Farjallah for the access to the RTM code.

This work has been carried out by the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, UVSQ, and by the PRiSM laboratory, thanks to the support of the French Ministry for Economy, Industry, and Employment throught the PERFCLOUD project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

References

1. Acumem: Acumem threadspotter. http://www.roguewave.com/products/threadspotter.aspx
2. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: tools for performance analysis of optimized parallel programs. http://hpctoolkit.org. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010). http://dx.doi.org/10.1002/cpe.v22:6
3. Alam, S.R., Barrett, R.F., Kuehn, J.A., Roth, P.C., Vetter, J.S.: Characterization of scientific workloads on systems with multi-core processors. In: IISWC, San Jose, pp. 225–236 (2006)
4. Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance tuning of x86 OpenMP codes with MAQAO. In: Parallel Tools Workshop, Dresden. Springer (2009)
5. Baysal, E., Kosloff, D., Sherwood, J.: Reverse time migration. Geophysics **48**, 1514–1524 (1983)
6. Beyler, J.C., Triquenaux, N., Palomares, V., Chabane, F., Fighiera, T., Halimi, J.P., Jalby, W.: MicroTools: automating program generation and performance measurement. In: ICPPW, Pittsburgh, pp. 424–433. IEEE (2012)
7. Burtscher, M., Kim, B.D., Diamond, J.R., McCalpin, J.D., Koesterke, L., Browne, J.C.: PerfExpert: an easy-to-use performance diagnosis tool for HPC applications. In: SC, New Orleans, pp. 1–11. IEEE (2010)
8. Charif-Rubial, A.S.: On code performance analysis and optimisation for multicore architectures. Ph.D. thesis (2012). http://tel.archives-ouvertes.fr/tel-00842601
9. Charif-Rubial, A.S., Barthou, D., Valensi, C., Shende, S.S., Malony, A.D., William Jalby, I.P.: MIL: a language to build program analysis tools through static binary instrumentation. In: HiPC'13, Hyderabad (2013)
10. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahàm, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture. In: STHEC, Kos, Greece (2008)
11. Gprof: The GNU profiler. http://sourceware.org/binutils/docs-2.18/gprof/index.html (2013)
12. Intel: Intel Vtune Amplifier XE. www.intel.com/software/products/vtune (2013)
13. Koliaï, S., Bendifallah, Z., Tribalat, M., Valensi, C., Acquaviva, J.T., Jalby, W.: Quantifying performance bottleneck cost through differential analysis. In: 27th ICS, Eugene, pp. 263–272. ACM, New York (2013). http://doi.acm.org/10.1145/2464996.2465440
14. Koliai, S., Zuckerman, S., Oseret, E., Ivascot, M., Moseley, T., Quang, D., Jalby, W.: A balanced approach to application performance tuning. In: LCPC, Newark, pp. 111–125 (2009)
15. Levon, J., Elie, P.: OProfile: a system profiler for Linux. http://oprofile.sourceforge.net (2013)
16. Liu, J., Yu, W., Wu, J., Buntinas, D., Kini, S., K, D., Wyckoff, P.: Microbenchmark performance comparison of high-speed cluster interconnects. IEEE Micro **24**, 42–51 (2004)
17. MAQAO: Maqao project. http://www.maqao.org (2013)
18. Martonosi, M., Gupta, A., Anderson, T.: MemSpy: analyzing memory system bottlenecks in programs. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Newport, pp. 1–12 (1992)
19. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: visualization and analysis of MPI resources. Supercomputer **12**, 69–80 (1996)

20. Real, F., Trumm, M., Vallet, V., Schimmelpfennig, B., Masella, M., Flament, J.P.: Quantum chemical and molecular dynamics study of the coordination of Th(IV) in aqueous solvent. J. Phys. Chem. B **114**(48), 15913–15924 (2010). http://dx.doi.org/10.1021/jp108061s
21. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). http://dx.doi.org/10.1177/1094342006064482
22. Sopeju, O., Burtscher, M., Rane, A., Browne, J.: AutoSCOPE: Automatic suggestions for code optimizations using PerfExpert. In: 2011 ICPDPTA, Las Vegas, Nevada, USA pp. 19–25 (2011)
23. Staelin, C.: lmbench: portable tools for performance analysis. In: USENIX Annual Technical Conference, San Diego, pp. 279–294 (1996)
24. Yoo, W., Larson, K., Kim, S., Ahn, W., Campbell, R.H., Baugh, L.: Automated fingerprinting of performance pathologies using performance monitoring units (PMUs). In: 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar'11), Berkeley, USENIX (2011)
25. Yoo, W., Larson, K., Baugh, L., Kim, S., Campbell, R.H.: ADP: automated diagnosis of performance pathologies using hardware events. In: Harrison, P.G., Arlitt, M.F., Casale, G. (eds.) SIGMETRICS, London, pp. 283–294. ACM (2012). http://dblp.uni-trier.de/db/conf/sigmetrics/sigmetrics2012.html#YooLBKC12